1

Model Checking, Abstraction, and Compositional Verification

David E. Long

July 1993

CMU-CS-93-178

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Edmund M. Clarke, Chair
Randal E. Bryant
Stephen D. Brookes
Orna Grumberg, The Technion

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, and in part by the National Science Foundation under Grant no. CCR-9005992, and in part by a National Science Foundation Graduate Fellowship, and in part by the Semiconductor Research Corporation under Contract 92-DJ-294.

Keywords: Formal verification, Temporal logic, Model checking, Compositional Verification, Abstraction, Mechanical verification, Futurebus+, Cache coherence, Protocol verifi cation

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

MODEL CHECKING, ABSTRACTION, AND COMPOSITIONAL VERIFICATION

DAVID LONG

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Edmund M. Clarke Jr — THESIS COMMITTEE CHAIR — July 9, 1993 — DATE

[illegible] — DEPARTMENT HEAD — 8/3/93 — DATE

APPROVED:

[illegible] — DEAN — 8/5/93 — DATE

Abstract

Because of the difficulty of adequately simulating large digital designs, there has been a recent surge of interest in formal verification, in which a mathematical model of the design is proved to satisfy a precise specification. Model checking is one formal verification technique. It consists of checking that a finite-state model of the design satisfies a specification given in temporal logic, which is a logic that can express properties involving the sequencing of events in time. One of the main drawbacks of model checking is the state explosion problem. This problem occurs in systems composed of multiple processes executing in parallel; the size of the state space generally grows exponentially with the number of components. This thesis considers two methods for avoiding the state explosion problem in the context of model checking: *compositional verification* and *abstraction*.

In compositional verification, our goal is to check local properties of the components in the design, deduce that these hold in the global system, and then use them to prove the overall specification. With abstraction, we can hide internal state, replace complex data types with simpler abstract ones, or simplify some of the timing behavior of the components. Using a connection between the abstracted and unabstracted systems, we deduce that whatever properties we prove at the abstract level also hold in the original system. We develop the necessary framework for using these two techniques with model checking, and demonstrate via a number of examples how they can be applied to realistic systems. Our largest example is the cache coherence protocol described in the IEEE Futurebus+ standard. In the course of the verification, we found errors in the standard, and proposed fixes for the protocol.

Acknowledgements

I have been very fortunate to have Ed Clarke as my advisor throughout my stay at Carnegie Mellon. Ed devotes considerable time and energy to his students, and many of my ideas about verification and about what constitutes interesting research arose from talks with him. His insistence on motivation, discussions, and examples has hopefully turned this thesis from a wasteland of formality into something readable. I was fortunate to be at CMU during the summers when Orna Grumberg visited. Her enthusiasm is contagious, and collaborations with her gave rise to many of the ideas presented here. Thanks also go to Randy Bryant and Steve Brookes, the other members of my committee. Their suggestions and comments have improved the thesis in numerous ways.

Much of my time at CMU was spent working with Jerry Burch and Ken McMillan. Observing them has contributed greatly to my development as a researcher. I especially enjoyed the countless talks with Jerry over dinner at Uncle Sam's Subs and The O. Had he not graduated and moved to California, I might never have had the motivation to finish.

The facilities staff gets a big thank-you for keeping all the machines in a very large department working smoothly. The administration in the School of Computer Science also deserves credit for keeping the creeping bureaucracy at bay. If I had been forced to deal with it, I would probably still be trying to fill out registration forms.

Finally, my parents and my sister gave me their constant love and encouragement; I can never thank them enough.

Contents

Chapter 1

Introduction

With society's increasing reliance on digital systems comes an increased emphasis on their dependability. Design errors can lead to serious failures, resulting in the loss of time, money, and, in some cases, lives. Further, even when an error is discovered during the design cycle, large amounts of effort can be required to correct the problem, especially if the error is found late in the process. For these reasons, we need methods that enable us to validate designs as early as possible. Traditionally, simulation has been the main debugging technique. However, because of the increasing complexity of digital systems, it is rapidly becoming impossible to simulate large designs adequately. For this reason, there has been a recent surge of interest in *formal verification*. In formal verification, a mathematical model of the design is compared with a formal specification describing the correctness criteria for the design. The verification is *exhaustive*: all possible behaviors of the model (and its environment) are considered. Further, the model of the system can be highly abstract, making it possible to check properties of a design during the earliest stages of its development.

Most formal verification methods fall into one of two classes. In *proof-based methods*, the designer constructs a mathematical proof, perhaps with the aid of some automated support, that the model meets its specification. Because the full power of mathematics is available, such techniques are very flexible. It is possible to model systems at almost any level of detail, and to prove properties of entire classes of systems. The main drawback of such methods is that they require a large amount of sophistication and effort on the part of the user. In contrast, *state-exploration methods* restrict the model to be finite-state and use state space search algorithms to check automatically that the specification is satisfied. Further, if the specification is false, then a *counterexample trace* can be produced to show the user why this is the case. This counterexample is invaluable in debugging the problem. The state-exploration methods require less expertise to use, but they do have some drawbacks. The most serious of these is the *state explosion problem.* This problem arises in systems composed of multiple components operating in parallel: the total number of states in the system generally grows exponentially with the number of components. This thesis is concerned with methods for attacking the state explosion problem.

The particular type of state-exploration method that we will be considering is called *temporal logic model checking.* Temporal logic is a logic for specifying how propositions change over time without introducing time explicitly [82]. It is a convenient formalism for specifying *reactive systems* (systems whose correct behavior is defined in terms of their interaction with an environment, rather than, e.g., their output upon termination) [75, 76]. In typical temporal logics, we have access to *temporal operators* such as "always" or "eventually". These operators can be nested, allowing us to express complex conditions. For example, we can specify that every time p is true, then at some later time q must be true by: "always, if p then eventually q". Temporal logic has been used extensively for specifying and verifying properties of hardware, starting with the work of Malachi and Owicki [66] and Bochmann [8], and most of our examples will be drawn from the area of computer hardware. Early verification was done by manual proofs, and as a result only very small systems could be checked. Further, the process was time-consuming and error-prone. (In fact, when Bochmann "verified" an arbiter design due to Seitz [83], he had to make some simplifying assumptions to make the proof manageable, and in the process, he missed a bug that was later found by Dill and Clarke [44].) The introduction of model checking procedures by Clarke and Emerson [27] and Quielle and Sifakis [80] was the first step towards being able to handle more realistic designs. In model checking, the design under consideration is described by a finite-state transition system, and an algorithm is used to verify that this system satisfies the specification. The use of model checking made it possible to find errors in nontrivial circuits which had been carefully designed [15, 44].

1.1 Scope of the Thesis

We discuss two main methods for avoiding the state explosion problem in the context of temporal logic model checking: *compositional verification* and *abstraction*. The goal of compositional verification is to try to take advantage of a given decomposition of the design into a number of components running in parallel. In our approach, this will mean that instead of forming the composition explicitly, we reason about small groups of components and then use the "local" properties that we verified to check the global specification. In abstraction, we try to simplify our models by hiding details. Verifying the simplified models is generally more efficient than checking properties of the original ones. When using abstraction, we must establish a relationship between the abstract models and the original ones, so that correctness at the abstract level will imply correctness for the original system. Abstraction can take many forms: we may hide parts of the system state, approximate complex data types with simpler ones, or simplify the temporal behavior of the design. In both cases, we are taking advantage of information about the design in order to simplify the verification task. Successful use of compositional verification requires some idea of how parts of the design contribute to satisfying the given specification. When using abstraction, we must balance the desire to hide information with the need to be able to prove the specification. This knowledge about the design must come from the person performing the verification.

The principle contributions of this thesis are as follows:

1. A method for constructing compositional verification systems using different types of temporal logic, and a particular compositional verification framework based on the logic CTL.

2. Methods for using abstraction within the above framework. We consider techniques for hiding state, abstracting data values, and abstracting complex timing behavior. We also develop ways of

efficiently producing the abstract models without explicitly constructing the unabstracted ones.

3. Ways of using *symbolic parameters* together with the above methods. Symbolic parameters essentially allow us to verify entire classes of properties or classes of systems simultaneously. In practice, the complexity of this verification is usually not much greater than the complexity of verifying an individual member of the class. We demonstrate how the use of symbolic parameters can greatly increase the power of our abstraction and compositional verification techniques.

4. Verification of part of the IEEE Futurebus+ standard [59]. We show that our techniques are practical by using them to verify the Futurebus+ cache coherence protocol. The verification is of independent interest as well, since we discovered errors in the IEEE standard.

1.2 Related Work

1.2.1 Temporal logic

There are a variety of temporal logic model checking procedures using a number of different logics [12, 27, 28, 33, 45, 64, 80, 92, 94, 95]. We will be concentrating on one particular logic, *CTL* [27], and one particular model of computation, but many of the ideas that we discuss are applicable to other logics and models. In contrast to our work, traditional model checking algorithms have dealt with the problem of determining whether a closed system satisfies a given specification. Part of our compositional verification framework is a *tableau construction* relating formulas in our logic with finite-state processes. It has a flavor similar to other tableau-like constructions [5, 24, 27, 46, 64, 78, 92, 95].

1.2.2 Efficient state-space search procedures

Much of the recent interest in formal verification methods has arisen from powerful techniques for searching large state spaces. By using *binary decision diagrams* (BDDs) (or more precisely, reduced, ordered BDDs) [11, 17, 18] to represent transition systems and state sets, it is possible to explore regular state spaces with extremely large numbers of states [4, 9, 22, 23, 24, 36, 37, 38, 47, 48, 67, 89]. *Partial-order approaches* attempt to cut down the search space by ignoring irrelevant interleavings of concurrent events in asynchronous systems [50, 67, 79, 90, 91]. All of these methods are useful for reducing the state explosion problem, but they are largely orthogonal to the methods that we consider. We do, however, make extensive use of the BDD-based techniques. They provide a powerful and flexible symbolic manipulation facility for working with sets and relations over finite domains. (A brief summary of BDDs is given in appendix A.)

1.2.3 Compositional verification

In this subsection, we survey methods designed to take advantage of the decomposition of a system into processes in order to simplify verification. *Local model checking algorithms* [33, 86, 94] based on logics like the *propositional μ-calculus* use a tableau-based procedure to deduce that a specific state (the initial state of the system) satisfies a given logical formula. The state space can be generated as needed in such an algorithm, and for some formulas, only a small portion of the space may have to be examined. Thus, by having a representation in terms of a set of components and producing global states only when required, it is sometimes possible to save significant time and space. The main drawback of these algorithms is that often the entire global state space is generated (for example, when checking that a property holds at every reachable state).

Winskel [93] proposes a method for decomposing logical specifications in the propositional μ-calculus into properties which the components of a system must satisfy for the specification to hold. The approach is appealing, but as might be expected, dealing with parallel composition is difficult. In our work, it is up to the user to derive appropriate specifications for the individual components.

Graf and Steffen [51] describe a method for generating a reduced version of the global state space given a description of how the system is structured and specifications of how the components interact.

Clarke, Long and McMillan [31, 32] describe a similar attempt. Both methods will still produce large state graphs if most of the states in the system are not equivalent, and much of the verification must be redone if part of the system changes. Shtadler and Grumberg [84] show how to verify networks of processes whose structure is described by grammars. In this approach, which involves finding the global behavior of each component, networks of arbitrary complexity can be verified by checking one representative system. For many systems, however, the number of states may still be prohibitive. While all of these methods do take advantage of the process structure, they are still constructing some form of a global state graph.

Compositionality is one of the main motivations behind the work on *process algebras* [7, 55, 57, 71]. By using equivalences or preorders, it is possible to construct hierarchical proofs of correctness of systems. At each stage, a small group of components is combined, internal actions are hidden, and the product is reduced. There are also links between the equivalences and preorders and various modal logics [55, 56]. One of our original approaches to compositional verification had much the same flavor: it was based on an equivalence between processes and a relationship between logical satisfaction and the equivalence [31].

Trace- and language-based methods [21, 43, 62] also support compositional verification. These methods are based on inclusion between sets of traces or sets of strings, and hence provide a natural framework for doing hierarchical correctness proofs. The approaches generally use linear-time semantics, while we will be concentrating on branching-time semantics and specifications in a temporal logic.

In 1984, Pnueli proposed the *assume-guarantee paradigm* for reasoning about concurrent systems [77]. In Pnueli's framework, we reason with triples of the form $\langle\varphi\rangle M\langle\psi\rangle$, where φ represents an assumption about the environment of M, and ψ is a guarantee about what will be true when this assumption holds. This approach is a powerful method for reasoning about concurrent systems. Pnueli used linear-time temporal logic (LTL) and a shared-memory process model. As most of our examples come from the hardware domain, this form of communication is not particularly appropriate. However, we would still like to be able to use the assume-guarantee paradigm. Our goal in chapters 2 and 3 will be to adapt the paradigm to a more traditional state-machine model.

We also demonstrate the practical value of the approach on a significant example, the Futurebus+ cache coherence protocol.

Josko [60] has developed a compositional verification methodology based on CTL. In his approach, specifications are given in a restricted form of CTL (essentially ACTL, as considered in section 2.5). Assumptions about the environment are given by a class of LTL formulas that are also expressible in ACTL. He gives an algorithm for checking whether a formula holds for a state machine given an assumption about the environment. The algorithm is based on labeling procedure that annotates states with subformulas of the specification and *derivatives* [19] of the assumption. The system does support assume-guarantee style reasoning. However, the algorithm is fairly *ad hoc*, the set of assumptions that can be expressed is restricted, and the method is not suitable for hierarchical verification or for using finite state induction techniques [63, 97]. Our approach does not suffer from these drawbacks.

Shurek and Grumberg [85] describe criteria for obtaining a compositional framework, and illustrate the idea using CTL* with only universal path quantifiers. This system is closest to the work presented in chapters 2 and 3. However, they give no provisions for handling fairness efficiently, using formulas as assumptions, or supporting temporal reasoning. For completeness purposes, models in their system are also associated with a fixed decomposition into components. Their overall focus is on proof systems and general aspects of modular verification, while ours is on demonstrating that these ideas are practical and can be used to simplify the verification of real systems.

1.2.4 Abstraction

Our main goal in using abstraction is to verify systems that manipulate data in nontrivial ways. Recently, symbolic model checking techniques [23, 24, 39, 67] have been used to handle circuits with data paths. The symbolic representations are able to capture much of the regularity in typical data manipulations. However, these methods are still unable to deal with some systems of realistic complexity. Our methods are designed to complement these techniques.

Wolper [96] has described how to use model checking to verify *data* *independent* systems. These are systems where the stored data values do not affect the course of the computation. For example, a protocol whose only function is to move data from a sender to a receiver (with no error checking, etc.) is typically data independent. Model checking for such systems can be done using only the control structure; the data can be abstracted away entirely. Unfortunately, many interesting systems are not data independent. In contrast, our techniques can cope with systems that are not data independent.

Van Aelten *et al.* [1] discuss a method for simplifying the verification of synchronous processors by abstracting away the data path. Their technique is to derive correctness conditions for the control circuitry by using a schedule of data path operations in the form of a signal flow graph (SFG). The data path is verified in a separate step. Claesen *et al.* [25] also discuss techniques for verifying digital signal processors against SFGs. These procedures are very specialized and efficient, but they cannot handle general properties: in a sense they just compare the control circuitry with the property specified by the SFG. Fujita [49] describes a method for verifying circuits with data paths by translating temporal logic specifications for the whole circuit into specifications involving only the control circuitry. In all of these approaches, dealing with feedback from the data path to the control circuitry is somewhat awkward. Corella [35] discusses a method for verifying circuits with data paths against algorithmic-level specifications. His approach involves constructing a state graph in which the data register values are terms built from variables and uninterpreted function symbols. The actual data path elements are verified separately. The method is not guaranteed to terminate, and it may give false negatives due to properties of the data path operations, but it has the advantage of being independent of data path width. It is not clear that it can be implemented using BDD-based representations, so it may not be able to handle circuits with complex control logic. Our use of symbolic parameters together with abstraction does not allow us to separate completely the control and data paths, but it does greatly simplify the verification. Further, our approach handles general properties and feedback from the data path to the control with ease.

Kurshan [62] did much of the pioneering work on using abstraction to verify finite-state systems. His approach has been automated in the COSPAN verification system [53, 54]. The basic notion of correctness is one of ω-language containment. Further, the user may use abstract models of the system and specification in order to reduce the complexity of the test for containment. To ensure soundness, the user specifies homomorphisms between the actual and abstract processes. These *homomorphic reductions* are checked automatically. Our work differs from Kurshan's in the following ways:

1. We are working in a branching-time rather than a linear-time framework. We concentrate on the use of temporal logics for specification.

2. The abstractions that we use correspond to language homomorphisms induced by boolean algebra homomorphisms in Kurshan's work. For this type of abstraction, we show how to derive automatically an *approximation* to the abstracted system. The approximation is constructed directly from a high-level representation of system (e.g., as a program in a finite-state language). It is not necessary to examine the state space of the unabstracted machine. Because of this, constructing the approximation is quite efficient. We demonstrate by example that this form of abstraction is powerful enough and that the approximation is accurate enough to allow us to verify interesting properties.

3. We show how to use symbolic parameters to increase the power of abstraction for verifying data-dependent systems.

General frameworks for abstraction are discussed by Burch [21] and by Bensalem *et al.* [6]. Burch's work is in the context of trace theory. He defines the notion of a *conservative approximation* between trace structures at different levels of abstraction. The approach of Kurshan can be viewed as a particular type of conservative approximation. Burch considers mainly applications to the verification of *real-time* systems. Bensalem *et al.* use the notion of a *Galois connection* between sets of states of two processes to define what it means for one process to be an abstraction of another. They also discuss the preservation of logical properties in the μ-calculus between abstract and concrete processes. The approach that we have chosen for formalizing our notion of abstraction is a type of cross between conservative approximations and

Galois connections. While both Burch and Bensalem *et al.* concentrate mainly on producing a theoretical framework, our emphasis is on efficiently producing abstract transition systems, combining abstraction with symbolic parameters, and demonstrating the application of these facilities to nontrivial examples.

The techniques that we use for efficiently producing abstract models from high-level representations are similar to those used in ***abstract interpretation*** [40, 41, 73, 74]. Abstract interpretation is a powerful method for program analysis that is based on constructing an abstract semantics for the programming language and then "executing" the program using these semantics. The semantics is designed so that this abstract execution always terminates. Abstract interpretation is used mainly to infer information that can help in generating more efficient code when compiling the program. As such, most abstract interpretations are designed to capture static information (e.g., what variables are live at this program point? are these two pointers ever aliased? is there a linear relation between these index variables?). When verifying reactive systems, it is the dynamic behavior of the system that is of interest. Further, abstract interpretations are generally constructed to collect a fixed type of information about programs in a fixed target language. In our work, the user has the flexibility to construct new abstractions dynamically and even to extend the description language. We then use symbolic manipulation techniques to produce automatically an appropriate abstract semantics.

Other techniques for producing reduced models have been proposed by Bouajjani *et al.* [10] and Dams, Grumberg and Gerth [42]. These approaches involve refining a partition of the set of states until a model which is minimal (in an appropriate sense) is obtained. While these procedures can make use of BDD-based representations for individual elements of the partition, the final result is essentially an explicit-state representation of the reduced model. Hence, when there are many behaviorally distinguishable states, these procedures may not be feasible. In contrast, our approach directly produces BDDs representing the abstract system.

Chapter 2

Compositional Verification, Part I

In this chapter, we consider methods for using *compositional model checking* to avoid the state explosion problem. The idea behind compositional methods is to exploit the natural decomposition of a system into communicating parallel processes. We will try to verify properties of individual components, infer that these properties hold in the complete system, and use them to deduce additional properties. The second step, inferring that local properties hold in the complete system, is the key requirement for compositional verification. Thus, we wish to examine the *compositional model checking problem*: how do we check that a specification is true of all systems that can be built using a given component? Below, we introduce the *temporal logic CTL*, show how it can be used to specify properties, and discuss the *Moore machine* model for finite state systems. We prove that the compositional model checking problem for full CTL is hard. Motivated by this result, we show that for a subset of CTL that we call ACTL, the problem is efficiently decidable. In subsequent chapters, we will use ACTL as the basis for doing full *assume-guarantee* style compositional reasoning and for using abstraction to simplify the verification of temporal properties.

2.1 CTL and Structures

Temporal logic is a logic for expressing the relative ordering of events in time without mentioning time explicitly. We will be using a temporal logic called *CTL* ("Computation Tree Logic") [27] as our basic specification formalism. Formulas in CTL are built up from:

1. atomic formulas, that express information about what is observable in a single system state;
2. the usual boolean connectives; and
3. temporal operators, that express how things change over time.

All temporal operators in CTL are interpreted relative to an implicit "current state", and each operator consists of two parts. The first is called a path quantifier and is either **A** or **E**. **A** denotes that something should be true of all "paths" (executions, expressed as sequences of states) starting at the current state. In contrast, **E** is used to specify the existence of a path with a certain property. The second part of a temporal operator is either **X**, **U**, or **V**. These are used to describe the ordering of events along the path or paths indicated by the **A** or **E**. The intuitive meanings of **X**, **U**, and **V** are as follows:

1. $\mathbf{X}\,\varphi$: **X** is read as "next time". $\mathbf{X}\,\varphi$ is true of a path if the formula φ is true at the second state on the path. Thus, **X** is used to express properties about the immediate successors of the current state.
2. $\varphi\,\mathbf{U}\,\psi$: **U** is the "until" operator. A path satisfies $\varphi\,\mathbf{U}\,\psi$ if:
   (a) there is some state on the path satisfying ψ; and
   (b) for all the preceding states, φ is true.

   Thus, φ is true up until a point where ψ is true.
3. $\varphi\,\mathbf{V}\,\psi$: The **V** operator is the dual of **U** and is read as "releases". A path satisfies $\varphi\,\mathbf{V}\,\psi$ if ψ is true at the current state, and ψ remains true up to and including the first point where φ is true. There is no requirement that φ ever become true, but when it does, it "releases" the requirement that ψ be true.

In a moment, we will look at some example specifications in CTL, but first, we give the formal definition of the class of CTL formulas. For the atomic formulas, we will assume that there is a set A of visible state components that we can observe. In a given state of our system, each component will have a specific value. We will assume that there is a set D_a of possible values for the state component a.

Definition 2.1 The logic *CTL* over a set of state components A is the set of formulas given by the following inductive definition:

1. The constant *true* is an atomic formula.
2. For each state component a in A and element d of D_a, $a = d$ is an atomic formula.
3. If φ and ψ are formulas, then $\neg\varphi$ and $\varphi \wedge \psi$ are formulas.
4. If φ and ψ are formulas, then $\mathbf{AX}\,\varphi$, $\mathbf{A}(\varphi\,\mathbf{V}\,\psi)$ and $\mathbf{A}(\varphi\,\mathbf{U}\,\psi)$ are formulas.

We will use the following abbreviations:

Abbreviation	Meaning
false	$\neg$*true*
$\varphi \vee \psi$	$\neg(\neg\varphi \wedge \neg\psi)$
$\varphi \rightarrow \psi$	$\neg\varphi \vee \psi$
$\varphi \oplus \psi$	$(\varphi \wedge \neg\psi) \vee (\neg\varphi \wedge \psi)$
$\varphi \Leftrightarrow \psi$	$\neg(\varphi \oplus \psi)$
$\mathbf{EX}\,\varphi$	$\neg\,\mathbf{AX}\,\neg\varphi$
$\mathbf{E}(\varphi\,\mathbf{U}\,\psi)$	$\neg\,\mathbf{A}(\neg\varphi\,\mathbf{V}\,\neg\psi)$
$\mathbf{E}(\varphi\,\mathbf{V}\,\psi)$	$\neg\,\mathbf{A}(\neg\varphi\,\mathbf{U}\,\neg\psi)$
$\mathbf{AG}\,\varphi$	$\mathbf{A}(\mathit{false}\,\mathbf{V}\,\varphi)$
$\mathbf{AF}\,\varphi$	$\mathbf{A}(\mathit{true}\,\mathbf{U}\,\varphi)$
$\mathbf{EG}\,\varphi$	$\mathbf{E}(\mathit{false}\,\mathbf{V}\,\varphi)$
$\mathbf{EF}\,\varphi$	$\mathbf{E}(\mathit{true}\,\mathbf{U}\,\varphi)$

Some of the operators are viewed as abbreviations for two reasons. First, by expressing $\mathbf{E}$ using the duality $\neg\,\mathbf{A}\,\neg$, we reduce the number of temporal operators that we have to consider when giving semantics or doing proofs. Second, certain patterns such as $\mathbf{A}(\mathit{true}\ \mathbf{U}$

φ) and $\mathbf{A}(\mathit{false}\ \mathbf{V}\ \varphi)$ occur often enough that it is convenient to have a special shorthand for them. **F** and **G** are intended to express eventuality and invariance respectively. $\mathbf{F}\,\varphi$ is true of a path when φ must hold at some state on the path (at some point in the "future"). $\mathbf{G}\,\varphi$ is true of a path when φ is true at every state on the path (is true "globally").

Let us now consider some example CTL formulas and their intuitive meanings.

1. $\mathbf{AG}(\mathit{req} = 1 \rightarrow \mathbf{AF}\,\mathit{ack} = 1)$: This formula states that for all reachable states (**AG**), if the state satisfies $\mathit{req} = 1$ ("a request is made"), then at some later point (**AF**) we must encounter a state with $\mathit{ack} = 1$ ("an acknowledgment is received"). Note that the **AF** is interpreted relative to the state where $\mathit{req} = 1$. The outer **AG** is interpreted starting with the initial states of the system.

2. $\mathbf{AG}\,\mathbf{AF}\,\mathit{enabled} = 1$: No matter what state we reach, at some later pointer we must encounter a state where $\mathit{enabled} = 1$. Note that after we pass a state where *enabled* is 1, then we must reach yet another such state. In other words, *enabled* must be 1 infinitely often.

3. $\mathbf{AG}\,\mathbf{EF}\,\mathit{restart} = 1$: For any reachable state, there must exist a path starting at that state that leads to a state satisfying $\mathit{restart} = 1$. It must always be possible to "restart the system".

Formally, CTL formulas are interpreted relative to a type of state transition system. The particular type of state transition system has traditionally been called a *Kripke structure*, after Kripke [2]. The only difference between our definition (below) and the traditional definition is that the visible state components in our transition systems may range over non-boolean domains. We will also abbreviate the name to just "structure".

Definition 2.2 A *structure* $M = \langle S, I, R, A, L\rangle$ is a tuple of the following form:

1. S is set of states.

2. $I \subseteq S$ is a set of initial states.

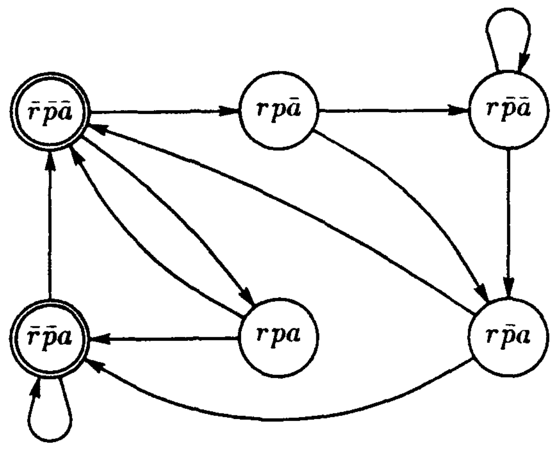


Figure 2.1: A structure

Next, we give the semantics of CTL relative to a structure. In the following definition, we use *comp*(φ) to denote the visible state components mentioned by the CTL formula φ. (The formal definition of *comp* is deferred.)

Definition 2.4 Let M be a structure and φ be a CTL formula with $A \supseteq comp(\varphi)$. *Satisfaction of φ by a state s of M*, denoted by $M, s \models \varphi$, is defined as follows:

1. $M, s \models true$.

2. $M, s \models a = d$ iff $L(s, a) = d$.

3. $M, s \models \neg\varphi$ iff it is not the case that $M, s \models \varphi$.
   $M, s \models \varphi \wedge \psi$ iff $M, s \models \varphi$ and $M, s \models \psi$.

4. Below, we use π to denote a sequence of states $s_0 s_1 s_2 \ldots$ from $s = s_0$.

   (a) $M, s \models \mathbf{AX}\,\varphi$ iff for every π, $M, s_1 \models \varphi$.

   (b) $M, s \models \mathbf{A}(\varphi\ \mathbf{U}\ \psi)$ iff for every π, there exists j such that $M, s_j \models \psi$ and for all $i < j$, $M, s_i \models \varphi$.

(c) $M, s \models \mathbf{A}(\varphi \mathbf{V} \psi)$ iff for all j, if φ is not satisfied at s_i for any $i < j$, then $M, s_j \models \psi$.

φ *is true of* M ($M \models \varphi$) if for every $s \in I$, $M, s \models \varphi$.

Later, we will occasionally have a need for *fixed point characterizations* of the CTL operators [27]. Suppose that S is a finite set of states and that F is a function mapping subsets of S to subsets of S. Also, assume that F is monotonic: if $S_1 \subseteq S_2$, then $F(S_1) \subseteq F(S_2)$. A *fixed point* of F is a set of states S_1 such that $S_1 = F(S_1)$. By Tarski's theorem [88], F has unique least and greatest fixed points (under the set inclusion ordering). The CTL operators involving **U** and **V** (and hence **F** and **G**) can be expressed as fixed points of an appropriate F. Below, we assume that all states in the structure have successors.

Consider, for example, a formula such as $\mathbf{A}(\varphi \mathbf{U} \psi)$. Let us assume that we know the sets of states S_φ and S_ψ where φ and ψ are true, respectively. A state will satisfy $\mathbf{A}(\varphi \mathbf{U} \psi)$ iff it either satisfies ψ immediately (is an element of S_ψ), or if it satisfies φ (is in S_φ) and all of its successors satisfy $\mathbf{A}(\varphi \mathbf{U} \psi)$. If we let $S_{\mathbf{A}(\varphi\mathbf{U}\psi)}$ denote the states satisfying $\mathbf{A}(\varphi \mathbf{U} \psi)$, then symbolically we have:

$$S_{\mathbf{A}(\varphi\mathbf{U}\psi)} = S_\psi \cup (S_\varphi \cap \mathbf{AX}\, S_{\mathbf{A}(\varphi\mathbf{U}\psi)}).$$

This suggests that $\mathbf{A}(\varphi \mathbf{U} \psi)$ can be expressed as a fixed point of the function

$$F(S_1) = S_\psi \cup (S_\varphi \cap \mathbf{AX}\, S_1).$$

In fact, the set of states satisfying $\mathbf{A}(\varphi\mathbf{U}\psi)$ is the least fixed point of this function. The least fixed point can be computed by starting with $\emptyset$ as an initial approximation and then repeatedly applying F. Eventually we will reach stability since the set of states is finite. Algorithmically, we begin with no states that are known to satisfy $\mathbf{A}(\varphi\mathbf{U}\psi)$. After applying F once, we obtain S_ψ as our approximation. At each successive step, any states that satisfy φ and whose successors are all known to satisfy $\mathbf{A}(\varphi \mathbf{U} \psi)$ will be added to the approximation.

Similarly, $\mathbf{A}(\varphi \mathbf{V} \psi)$ is the greatest fixed point of the function

$$F(S_1) = S_\psi \cap (S_\varphi \cup \mathbf{AX}\, S_1).$$

Fixed point characterizations for operators such as **AG** and **EG** can be derived by expressing these operators in terms of the ones above.

2.2 Moore Machines

We now consider one class of systems that we would like to verify: synchronous digital circuits. Such a circuit consists of a number of latches or state-holding registers, plus logic that updates these latches based on the current state of the system and inputs from the environment. There is a global clock, and during each clock cycle, the values in the latches and the inputs are used to drive the logic and compute the next state value of the system. One common model for systems such as this is the *Moore machine* [72]. A Moore machine is a kind of state transition system with distinct inputs and outputs. During each step of the computation of a Moore machine, the environment supplies an input, the machine makes a transition, and, based on the final state, gives an output. The formal definition is as follows.

Definition 2.5 A *Moore machine* $M = \langle S, I, A_I, A_O, R, L\rangle$ is a tuple of the following form:

1. S is a set of states.

2. $I \subseteq S$ is a nonempty set of initial states.

3. A_I is a set of input state components. Each element a of A_I has a corresponding domain D_a of possible values.

4. A_O is a set of output state components. Each element a of A_O has a corresponding domain D_a of possible values.

5. R is a transition relation, relating a starting state in S, a labeling function over A_I, and an ending state in S. For every $s_0 \in S$ and labeling function f over A_I, there must exist some $s_1 \in S$ such that $R(s_0, f, s_1)$.

6. L is a function that takes a state and an output state component a and returns an element of D_a.

The sets of input and output state components must be disjoint.

Note that we allow our Moore machines to be nondeterministic. That is, for one particular input, we may have transitions to two states with the same output labeling. Synchronous circuits are deterministic, but we often want to use nondeterminism in modeling. As we will see in later examples, nondeterminism allows us to:

1. model classes of circuits or incompletely specified designs; and
2. hide internal state and simplify the verification process.

Example 2.2 The circuit shown in figure 2.2 is an implementation of the protocol described in example 2.1. It consists of two registers, r and p, and has one input a. The initial value in the registers is assumed to be logic 0. The Moore machine corresponding to this circuit

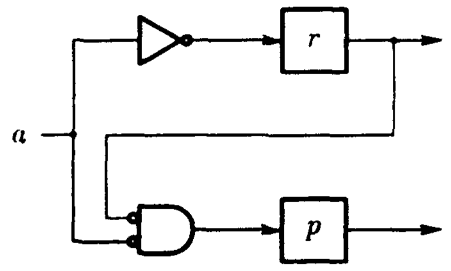


Figure 2.2: A handshake circuit

is shown in figure 2.3. The state labelings and initial states are indicated as in our earlier example. Conditions on the arcs are used to give the input conditions under which the transition can be taken. □

Moore machines that have disjoint sets of output state components can be composed in a natural way. In a composition of two Moore machines, each machine may receive some of its inputs from the other element of the composition and some of its inputs from the (as yet unspecified) environment. The composed machine has as outputs all of the outputs of the components. Its inputs are all those inputs that are not tied to outputs from other components during the composition. At the circuit level, Moore machine composition corresponds to wiring outputs from each machine to appropriate inputs of the other.

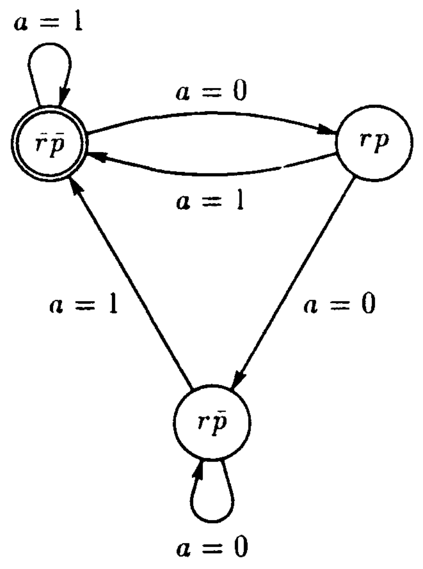


Figure 2.3: Moore machine for the circuit of figure 2.2

Example 2.3 The circuit shown in figure 2.4 is a possible environment for the circuit of example 2.2. It receives requests via the input r and gives acknowledgments using the output a. It also has an output q that becomes 1 when it first produces an acknowledgment. When we compose the two circuits, the r output of the circuit in figure 2.2 is tied to the input r of the circuit in figure 2.4. Similarly, the output a of the circuit in figure 2.4 drives the a input of the circuit in figure 2.2. The overall circuit is shown in figure 2.5. □

Definition 2.6 The *composition of Moore machines* M and M' (denoted $M \parallel M'$) is defined when $A_O \cap A'_O = \emptyset$ and is then the Moore machine M'' defined by:

1. $S'' = S \times S'$.
2. $I'' = I \times I'$.
3. $A''_I = (A_I - A'_O) \cup (A'_I - A_O)$.
4. $A''_O = A_O \cup A'_O$.

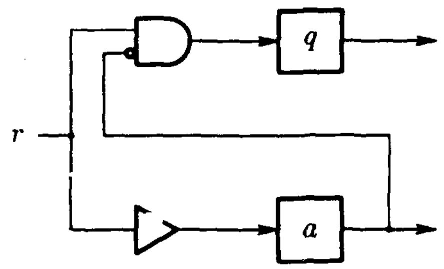


Figure 2.4: Environment for the circuit of figure 2.2

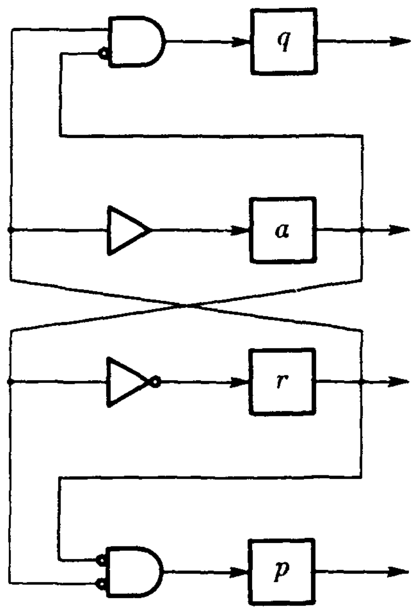


Figure 2.5: Composed circuit

5. $R''((s_0, s'_0), f'', (s_1, s'_1))$ iff $R(s_0, f, s_1)$ and $R'(s'_0, f', s'_1)$, where $f = f'' \cup (L'(s'_0) \downarrow A_I)$ and $f' = f'' \cup (L(s_0) \downarrow A'_I)$. The idea here is to say that:

   (a) each machine must take a step; and

   (b) the inputs that each machine sees are the inputs from the overall environment plus the outputs from the other machine in the composition.

   We are using $\cup$ and $\downarrow$ to denote enlarging and restricting the domain of a labeling function. $L'(s'_0) \downarrow A_I$ is the labeling function whose domain is $\mathit{dom}(L'(s'_0)) \cap A_I$ and which agrees with $L'(s'_0)$ on this set. In other words, it represents the outputs of M' that M is going to observe. $f'' \cup (L'(s'_0) \downarrow A_I)$ is the labeling function with domain $\mathit{dom}(f'') \cup (\mathit{dom}(L'(s'_0)) \downarrow A_I)$ that agrees with f'' on $\mathit{dom}(f'')$ and with $L'(s'_0)$ on $\mathit{dom}(L'(s'_0)) \downarrow A_I$. Thus, it represents all the inputs to M: those from the external environment (f'') and those from M' ($L'(s'_0) \downarrow A_I$).

6. $L''((s, s')) = L(s) \cup L'(s')$.

Example 2.4 The Moore machine for the circuit of figure 2.4 is shown in figure 2.6. Composing this Moore machine with the Moore machine of figure 2.3 yields the Moore machine shown in figure 2.7. (Here we are showing only the reachable states of the composition.) On examining the result of the composition, we see that it does in fact represent the composite circuit (figure 2.5). □

2.3 Moore Machines and CTL

We now have two models of computation: structures and Moore machines. We also have a temporal logic, CTL, whose semantics are defined over the former. In this section, we consider the question of how to define the semantics of CTL for Moore machines. Recall our previous circuit example. In our composed circuit (figure 2.5), there are no free inputs. As a result, the Moore machine for this circuit (figure 2.7) looks very much like a structure. Also, we have the intuition

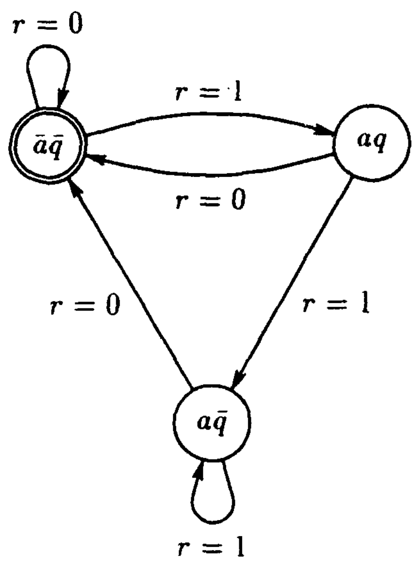


Figure 2.6: Moore machine for the circuit of figure 2.4

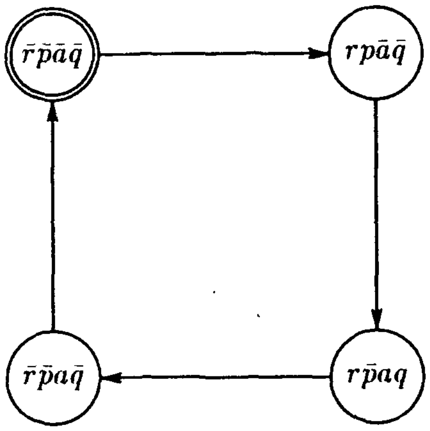


Figure 2.7: Composition of Moore machines

that the behavior of the circuit cannot be altered by connecting it to other circuits. Thus, it seems natural to define a correspondence between Moore machines with no free inputs and structures. (In fact, due to the isomorphism between such Moore machines and structures, we will sometimes identify them for notational convenience.) Then, we will define the semantics of CTL for these Moore machines by using the corresponding structure.

Definition 2.7 A Moore machine M is called *closed* if it has no free inputs, i.e., if $A_I = \emptyset$. A *closing environment* for M is a Moore machine M'' with $A_O \cap A''_O = \emptyset$ and $A_I \subseteq A''_O$. Thus, M and M'' can be composed, and the result will be closed.

Definition 2.8 The *structure M' for the Moore machine M* (denoted $struct(M)$) is defined as follows:

1. $S' = S \times labelings(A_I)$. (Recall that $labelings(A_I)$ is the set of all labeling functions over A_I.)
2. $I' = I \times labelings(A_I)$.
3. $R'((s_0, f_0), (s_1, f_1))$ iff $R(s_0, f_0, s_1)$.
4. $A = A_I \cup A_O$.
5. $L'((s, f), a) = f(a)$ for $a \in A_I$. $L'((s, f), a) = L(s, a)$ for $a \in A_O$.

In the above definition, we actually assign a structure to an arbitrary Moore machine, not just a closed one. The reason for this will become clear later; for now, assume that the Moore machine M above is closed. Now we define satisfaction in terms of $struct(M)$.

Definition 2.9 Let M be a closed Moore machine, and let φ be a CTL formula with $A_O \supseteq comp(\varphi)$. Then $M \models \varphi$ iff $struct(M) \models \varphi$.

(Note that the fact that a Moore machine is closed does not mean that it cannot be composed with other Moore machines. Given this, there needs to be some argument that such compositions do not affect the closed machine in any real way. For now, we just state that this is indeed the case: given a closed machine M, a formula φ, and a closing environment M', we have $M \models \varphi$ iff $M \parallel M' \models \varphi$. The proof of this is deferred.)

Let us now consider non-closed Moore machines. One possible way to define the semantics of CTL for such machines is to just assume that the environment can give any input at any point. With this assumption, we can produce a structure for an arbitrary machine M. The idea will be as follows: each state of M will be split into a number of structure states, one for each possible input that the environment could give. The transitions out of one of the structure states s are determined by looking at the Moore machine transition relation and seeing which transitions are enabled given the particular input represented by s. In fact, the structure obtained in this way is exactly *struct*(M) as defined above. Now we can again just take $M \models \varphi$ iff *struct*$(M) \models \varphi$. This is the approach that has traditionally been used [13, 14].

Example 2.5 Consider the non-closed Moore machine of example 2.2. This Moore machine, shown in figure 2.3, is represented by the structure given in figure 2.1. Each state of the Moore machine has been split into two structure states, one for the case when $a = 0$ and one for the case when $a = 1$. □

With this definition of when a CTL formula is true for a Moore machine, we have that the machine of figure 2.3 satisfies the formula

$$\mathbf{AG}(r = 1 \wedge p = 1 \wedge a = 0 \rightarrow \mathbf{EX}\,\mathbf{EX}(r = 1 \wedge p = 0)).$$

Note, however, that when we compose this Moore machine with the one in figure 2.6 (obtaining the Moore machine in figure 2.7), the formula ceases to hold. On the other hand, according to this definition, the machine of figure 2.3 also satisfies

$$\begin{aligned}&\mathbf{EF}(a = 0 \wedge \mathbf{EX}(a = 0 \wedge \mathbf{EX}\,a = 0))\\ &\qquad \rightarrow \mathbf{EF}(r = 1 \wedge p = 1 \wedge a = 0 \wedge \mathbf{EX}\,\mathbf{EX}(r = 1 \wedge p = 0)).\end{aligned}$$

("If it is possible for three $a = 0$ inputs to occur in a row, then it is also possible to pass through the state $rp\bar{a}$ and to be in one of the states $r\bar{p}a$ or $r\bar{p}\bar{a}$ in two more steps.") This formula in fact remains true no matter what closing environment we use for the machine. *In order to be able to do compositional reasoning, we must have some way of distinguishing between these two situations.* That is, we need to be able to tell when a formula is true of *all* possible closed systems that we could build using a given non-closed machine. Motivated by this requirement, we now give the definition of satisfaction of a formula that we will use from this point on.

Definition 2.10 Let M be a Moore machine, and let φ be a formula with $A_I \cup A_O \supseteq comp(\varphi)$. We say that M *satisfies* φ ($M \models \varphi$) when for every closing environment M' for M, $struct(M \parallel M') \models \varphi$.

Obviously, this is not a definition that immediately suggests any procedure for checking whether $M \models \varphi$. The problem of deciding, for a particular class of formulas, whether or not a Moore machine satisfies a formula in that class will be called the *compositional model checking problem* for the class. In the remainder of this chapter, we first show that there is probably no efficient algorithm for the compositional model checking problem for full CTL. However, we will show that for a subset of the logic called ACTL, the problem *is* efficiently decidable. This result will serve as the basis for the remainder of the thesis: using ACTL, we give methods for doing full *assume-guarantee* style reasoning and for using *abstraction* to simplify the verification process.

Before proceeding, we must say a word about what we consider to be an efficient algorithm. Consider a Moore machine M where A_I is a set of input components ranging over $\{0, 1\}$ and A_O is empty. Also, suppose $S = I = \{s_0\}$ and that there is a transition from s_0 to s_0 on any input. The traditional model checking algorithm for CTL on Moore machines has complexity $\Omega(2^{|A_I|})$ in this case, even for purely propositional formulas. This is precisely because each state of M is viewed as being represented by $2^{|A_I|}$ structure states. In fact, for a purely propositional formula φ, checking whether $M \models \varphi$ is equivalent to checking whether φ is a tautology. Given this observation, we cannot expect to obtain an algorithm that runs in time subexponential in $|A_I|$. Thus, we will consider an algorithm that is exponential in $|A_I|$ but polynomial in $|S|$, $|R|$, $|\varphi|$, etc., to be "efficient".

2.4 Compositional Verification and CTL

In this section, we consider the compositional model checking problem for full CTL: given a Moore machine and a CTL formula, decide whether the formula is true of all closed systems containing the Moore machine. We show that there is probably no efficient algorithm to solve it. More specifically, we prove that even if M is represented by its corresponding structure (i.e., the input is already exponential in $|A_I|$), then the compositional model checking problem for CTL is still NP-hard.

The reduction will be from 3SAT [34]. Let $f = c_0 \wedge c_1 \wedge \cdots \wedge c_{m-1}$ be a 3SAT formula, and let the variables in f be x_0, x_1, ..., x_{n-1}. We are going to construct a Moore machine M that will receive a sequence of inputs, one per variable of f, denoting whether each variable is true or false. Given such a sequence of inputs, the terminal reachable states of M will indicate whether each conjunct in f is true or false for those particular variable values and so tell whether f is satisfied. The quantification over all closing environments is used to quantify over all possible input sequences, i.e., all valuations of the x_j.

Conceptually, the inputs to M will take on values from the set

$$\{x_0, \neg x_0, \ldots, x_{n-1}, \neg x_{n-1}\}.$$

We encode these possible inputs using $\lceil \log_2 2n \rceil$ boolean input state components. The input sequence representing the valuation for the x_j will be of the form i, i_0, ..., i_{n-1}, ..., where i is an arbitrary initialization input, i_j is either x_j or $\neg x_j$, and the inputs after i_{n-1} are arbitrary. Conceptually, the output labeling function for each state will denote one of the values

$$\{\mathit{nothing}, c_0, \neg c_0, \ldots, c_{m-1}, \neg c_{m-1}\}.$$

These are encoded with $\lceil \log_2(2m+1) \rceil$ boolean output state components. For clarity, when writing inputs and outputs, we will use the conceptual values above. Also, when labeling states in a figure, we use no label to indicate *nothing*.

Let e_j denote either x_j or $\neg x_j$, and let $\neg e_j$ be $\neg x_j$ if $e_j = x_j$ and x_j if $e_j = \neg x_j$. For each conjunct $c_k = (e_{j_0} \vee e_{j_1} \vee e_{j_2})$ of f, we construct a recognizer that will tell whether the conjunct is satisfied. Assume without loss of generality that $j_0 < j_1 < j_2$. The recognizer for this conjunct is shown in figure 2.8. Obviously, given a sequence of inputs as described above, this recognizer will reach the state labeled c_k if the conjunct evaluates to true and will reach the state labeled $\neg c_k$ otherwise. Also, once it reaches either of these states, it remains there regardless of any further inputs.

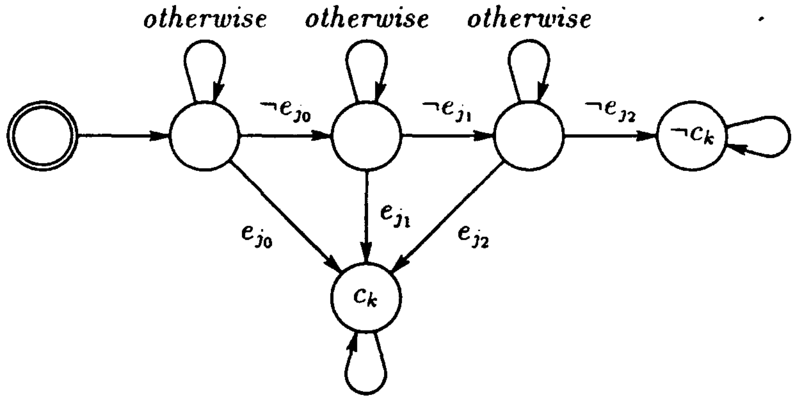


Figure 2.8: Recognizer Moore machine for a conjunct

The Moore machine M will consist of a group of recognizers, one per conjunct. These recognizers all share their initial state, i.e., M has exactly one initial state. Consider an environment which supplies a sequence of inputs of the form described earlier to M. In this environment, the state labeled $\neg c_k$ is reachable iff the corresponding valuation of the x_j makes the conjunct c_k false. Thus, the valuation represented by the environment makes f false iff for some k, there exists a path in the composition of M and the environment to a state labeled $\neg c_k$. Based on this, it is tempting to suggest that f is satisfiable iff it is not the case that every closed system containing M satisfies the formula

$$(\mathbf{EF}\,\neg c_0) \vee (\mathbf{EF}\,\neg c_1) \vee \cdots \vee (\mathbf{EF}\,\neg c_{m-1}).$$

This is not quite the case however, in that any arbitrary environment may not behave as we would like. For example, it may never give an input signaling the truth value of some particular x_j, or it may give an input saying that x_j is true and then later give an input saying that it is false. One way to try to exclude this type of behavior would be to add a kind of "syntax checker" to M, such as the one shown in figure 2.9 (in the figure, "*ow*" denotes "otherwise"). However, this leads to complications if the environment nondeterministically chooses different variable values on different paths.

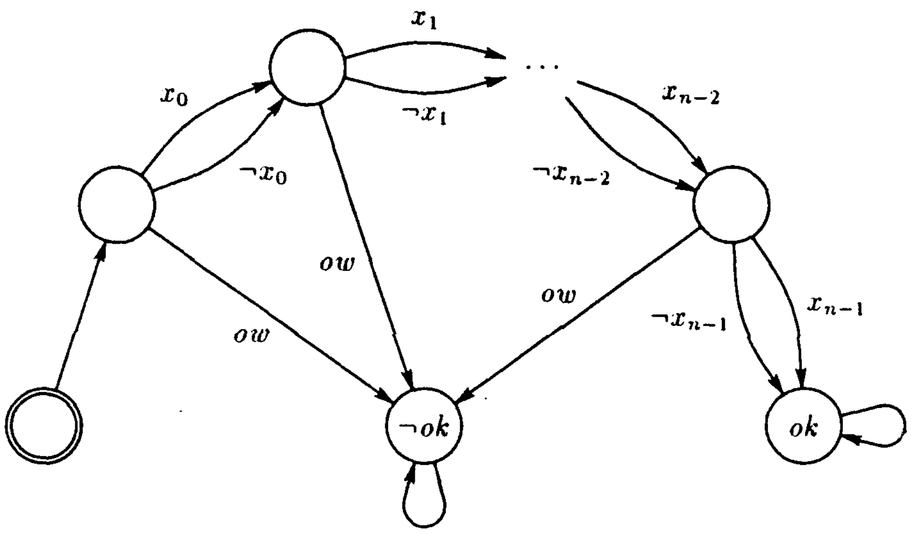


Figure 2.9: Syntax checking Moore machine

Instead of adding such a checker, we modify our CTL formula. After one arbitrary input, the environment may either supply an x_0 or a $\neg x_0$, but it may not output anything else, nor do we want it to nondeterministically choose different values on different paths. That is, it must satisfy

$$(\mathbf{AX}\, x_0) \lor (\mathbf{AX}\, \neg x_0).$$

In general, after $j+1$ steps, it must supply a unique value for x_j, and hence must satisfy

$$(\mathbf{AX}^{j+1}\, x_j) \lor (\mathbf{AX}^{j+1}\, \neg x_j),$$

where $\mathbf{AX}^j\, p$ is an abbreviation for

$$\overbrace{\mathbf{AX}\,\mathbf{AX}\ldots\mathbf{AX}}^{j}\,p.$$

This leads us to the desired result (the proof is deferred).

Theorem 2.1 f is satisfiable iff it is not the case that every closed system containing M satisfies the formula

$$\bigwedge_{j=0}^{n-1}((\mathbf{AX}^{j+1}\,x_j) \vee (\mathbf{AX}^{j+1}\,\neg x_j)) \rightarrow \bigvee_{k=0}^{m-1} \mathbf{EF}\,\neg c_k.$$

To complete the argument that the compositional model checking problem for CTL is NP-hard even when the Moore machine is given as a structure, we need to show that the Moore machine constructed above can be constructed in time polynomial in the size of f. Obviously it will be enough to observe that the (structure for the) Moore machine has size polynomial in the size of f. The Moore machine has m recognizers, each of which has six (Moore machine) states. The state labeling for each state uses $O(\log_2 m)$ bits. The input encoding is $O(\log_2 n)$ bits long, where n is the number of variables appearing in f. Hence when we expand the Moore machine into a structure, we get a factor of n increase in size. Overall, the number of bits needed to represent the states of the Moore machine is $O(mn \log_2 m)$. The number of bits needed to represent the transitions is at worst $O(n(mn \log_2 m)^2)$.

Before moving on, we note that we can obtain an efficient *approximation algorithm* for the compositional model checking problem for full CTL. Consider why the compositional model checking problem for CTL is difficult. First, it is generally not possible to decompose a formula into subformulas, check the subformulas, and combine the results. For example, consider checking $\mathbf{EX}(a = 1) \vee \mathbf{EX}(a = 0)$ on a Moore machine where a is an input ranging over $\{0, 1\}$. Obviously, the formula as a whole will be true regardless of what the environment does. However, $\mathbf{EX}\, a = 1$ is certainly not true for all environments, nor is $\mathbf{EX}\, a = 0$. Thus, determining whether the two subformulas are true in all environments does not help us solve the overall problem.

A related difficultly arises in situations such as the one shown in figure 2.10. Consider trying to determine whether $\mathbf{EX}\,\mathbf{EX}\,b = 1$ is true of all systems containing the Moore machine shown in the figure. In the standard CTL model checking algorithm, we would use the truth value for $\mathbf{EX}\,b = 1$ at the two successors of the initial states to determine whether $\mathbf{EX}\,\mathbf{EX}\,b = 1$ was true at the initial state. For this example, there are environments that make $\mathbf{EX}\,b = 1$ false at the left successor and others that make the formula false at the right successor. However, the overall formula is in fact true in all environments. This is because no environment can distinguish between the two successors based on their labeling. Hence, if the environment supplies the input $a = 1$ to the left successor, $b = 1$ becomes true in the next state. If it supplies only $a = 0$, then it must also supply $a = 0$ to the right successor, and this will again lead to a state where b is true. Thus we cannot just look at immediate successors when evaluating temporal operators.

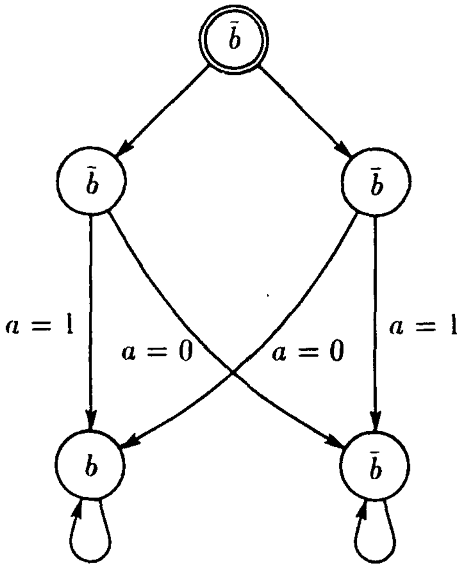


Figure 2.10: A nondeterministic Moore machine

Our approximation algorithm will be designed to avoid these problems. Given a formula and a Moore machine M, the algorithm will indicate either:

1. the formula is true of all closed systems containing M; or
2. the formula is false of all closed systems containing M; or
3. the truth value of the formula for all closed systems contain M is unknown.

Our approximation algorithm will be efficient, but it will not be able to resolve all difficult situations such as those discussed above.

The basic idea behind the algorithm will be to separate out the branching in the environment (input nondeterminism) from the branching in the Moore machine itself (internal nondeterminism). When checking a formula such as $\mathbf{EX}\,\varphi$ at a state, we will see whether for all input choices, there exists an internal choice such that we reach a state where φ must hold. The basic structure of the algorithm will then be similar to standard CTL model checking methods. We proceed in a bottom-up fashion, starting at the atomic subformulas and working our way towards the top-level formula. Operators such as **EF** will be evaluated using fixed point techniques. The full approximation algorithm and a proof of its correctness is deferred until the end of the chapter.

2.5 ACTL

In this section, we show that there is a subset of CTL, which we call ACTL [30, 52, 60, 85], for which the compositional model checking problem is efficiently decidable. Further, this subset is sufficiently expressive to cover almost all of the temporal formulas that are used as specifications in practice. The basic idea behind ACTL is to eliminate the ability to talk about the existence of a path, i.e., the **E** path quantifier. Once the logic can only talk about behavior over all paths, we will just need to consider a single "maximal" closing environment in order to solve the compositional model checking problem. Intuitively, composing with any other closing environment will eliminate some paths, and since our formulas only talk about behavior over all paths, such pruning will not change a formula from true to false. Further, if the composition of the given component with its maximal closing environment does not satisfy the specified formula, then the formula obviously cannot be true of all closed systems containing the component. We begin by formally defining ACTL; in order to ensure that **E** does not arise via duality, we require that formulas be in a type of negation-normal form. Thus, negations can only be applied to atomic formulas.

Definition 2.11 The logic *ACTL* over a set of state components A is the set of formulas given by the following inductive definition:

1. The constant *true* is an atomic formula.
2. For each state component a in A and element d of D_a, $a = d$ is an atomic formula.
3. If φ is an *atomic* formula, then $\neg\varphi$ is a formula.
4. If φ and ψ are formulas, then $\varphi \wedge \psi$ and $\varphi \vee \psi$ are formulas.
5. If φ and ψ are formulas, then $\mathbf{AX}\,\varphi$, $\mathbf{A}(\varphi\,\mathbf{V}\,\psi)$ and $\mathbf{A}(\varphi\,\mathbf{U}\,\psi)$ are formulas.

We may sometimes write an ACTL property using **E**; in these cases, pushing negations inwards using duality will result in a proper ACTL formula.

ACTL is sufficient to express many interesting properties. In fact, almost all CTL specifications that are used in practice are expressible in ACTL. Intuitively, this is because we generally want to require that a system *must* behave correctly, rather than that it *may* behave correctly. The most commonly used CTL properties that cannot be expressed in ACTL are those describing weak progress requirements. As an example, the formula $\mathbf{AG}\,\mathbf{EF}\,restart = 1$ that we mentioned earlier is not expressible in ACTL.

We now show that the compositional model checking problem for ACTL is efficiently decidable. To do this, we will prove that it is enough to consider the composition of the component M with the following environment when doing the model checking.

Definition 2.12 The *maximal closing environment* for the Moore machine M, denoted $E(M)$, is the Moore machine M' defined as follows:

1. $S' = F$, where F is the set of all labeling functions over A_I.
2. $I' = F$.
3. $A'_I = \emptyset$.
4. $A'_O = A_I$.
5. $R'(s'_0, f', s'_1)$ is identically true.
6. $L'(f, a) = f(a)$

Example 2.6 The maximal closing environment for the Moore machine of figure 2.3 is shown in figure 2.11. It has no inputs and one output, a, corresponding to the inputs of the Moore machine in the earlier figure. Composing the Moore machine of figure 2.3 with its maximal environment gives the result shown in figure 2.12. □

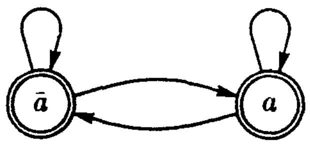


Figure 2.11: The maximal closing environment for the Moore machine of figure 2.3

The reader may think that the state diagram in figure 2.12 looks familiar. In fact, it is the same as the one in figure 2.1, which happens to be the structure for the Moore machine of figure 2.3. In general, the composition of the Moore machine M together with $E(M)$ gives a Moore machine that is isomorphic to *struct*(M). Thus, when checking whether *struct*$(M \parallel E(M)) \models \varphi$, we are essentially just checking that *struct*$(M) \models \varphi$. This means that doing the composition with the maximal closing environment does not really increase the size of the state graph that we are working with. We now turn to the main result of this section.

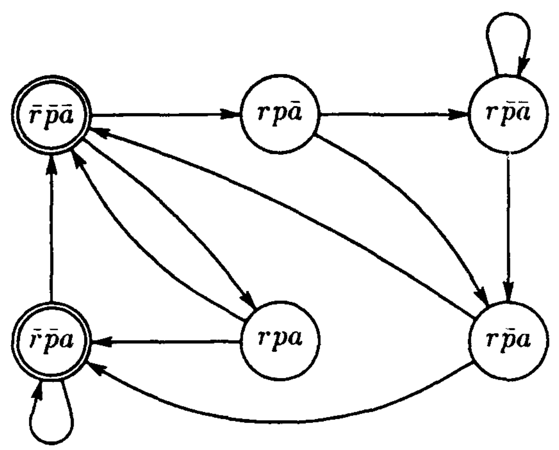


Figure 2.12: The composition of the Moore machine of figure 2.3 with its maximal closing environment

Theorem 2.2 Let M be an arbitrary Moore machine, and suppose that φ is an ACTL formula with $A_I \cup A_O \supseteq comp(\varphi)$. Then $M \models \varphi$ iff $struct(M \parallel E(M)) \models \varphi$.

The formal proof of this is deferred until the end of the chapter; here, we just try to give the intuition why it is true. The key idea is to note that if M' is a closing environment for M, then there is a natural mapping from states of $M \parallel M'$ to states of $M \parallel E(M)$. To see this, consider a state s' of M'. Since M' is a closing environment for M, the output labeling of s' must give values to all the state components in A_I. Hence, we can view s' as giving rise to a labeling function over A_I. However, each such labeling function is a state of $E(M)$, and so for each s', we have a corresponding state $s'_{E(M)}$ of $E(M)$. Now a state (s, s') of $M \parallel M'$ will just be identified with $(s, s'_{E(M)})$ in $M \parallel E(M)$.

Example 2.7 Let M be the Moore machine of figure 2.3. Recall that the composition of M with its maximal closing environment (figure 2.11) is given by figure 2.12. Now let M' be the Moore machine of figure 2.6. M' is a closing environment for M, and the composition of M and M' is shown in figure 2.7. For each state in $M \parallel M'$, we can obtain a corresponding state in $M \parallel E(M)$ by dropping the labeling for the state component q. As an example, the state $\bar{r}\bar{p}a\bar{q}$ in $M \parallel M'$ maps to the state $\bar{r}\bar{p}a$ in $M \parallel E(M)$. $\square$

Further, the mapping above also has two nice properties:

1. initial states of $M \parallel M'$ map to initial states of $M \parallel E(M)$; and
2. pairs of states, i.e., transitions, of $M \parallel M'$ map to transitions of $M \parallel E(M)$.

In essence, the Moore machine $M \parallel M'$ can be embedded in the Moore machine $M \parallel E(M)$. Now consider a formula of ACTL. The formula describes properties of all paths from a state. If such a formula is false at some state in $M \parallel M'$, then we can find some path demonstrating why it is false. This path is then mapped into a corresponding path in $M \parallel E(M)$. By using an inductive argument, we can prove that this path demonstrates that the corresponding state in $M \parallel E(M)$ does not satisfy the property either. This argument shows that if we verify that a formula is true for $M \parallel E(M)$, then we know that the formula holds in all closed systems that contain M. Further, if the formula is false for $M \parallel E(M)$, then obviously we have found a closed system containing M for which the formula is false.

2.6 Summary

We have considered the issues involved in using the temporal logic CTL to specify properties of systems of Moore machines. The desire to do compositional reasoning led us to consider the compositional model checking problem: given a Moore machine and a formula, is the formula true in all closed systems that can be built using the Moore machine. We showed that there is probably no efficient algorithm for solving this problem in the case of general CTL formulas. However, we also proved that the problem can be solved efficiently for the CTL subset ACTL. ACTL will be used in the following chapters as the basis for doing assume-guarantee style reasoning and for using abstraction. The remainder of this chapter is devoted to filling in some of the formal details and proofs that were deferred earlier.

2.7 Technical Details

First, here is the formal definition of *comp*, the function that returns the set of state components that appear in a formula.

Definition 2.13 The set $comp(\varphi)$ of state components of the formula φ is defined as follows:

1. $comp(true) = \emptyset$.
2. $comp(a = d) = \{a\}$.
3. $comp(\neg\varphi) = comp(\varphi)$. $comp(\varphi \wedge \psi) = comp(\varphi) \cup comp(\psi)$.
4. $comp(\mathbf{AX}\,\varphi) = comp(\varphi)$.
   $comp(\mathbf{A}(\varphi\,\mathbf{U}\,\psi)) = comp(\varphi) \cup comp(\psi)$.
   $comp(\mathbf{A}(\varphi\,\mathbf{V}\,\psi)) = comp(\varphi) \cup comp(\psi)$.

Now let us go back to the definition of satisfaction of a formula by a closed Moore machine (definition 2.9). We remarked there that since closed Moore machines can still be composed with other Moore machines, there needed to be an argument that such composition did not really affect the closed Moore machine. This notion will be formalized using a notion of *bisimulation equivalence* [71] between Moore machines. The basic idea will be to show that if we have a closed Moore machine M, and we compose M with a closing environment M', then M and $M \parallel M'$ will be equivalent. We will then appeal to the well-known result that equivalent structures satisfy the same CTL formulas [16]. There is one detail that we must take care of first however: M and $M \parallel M'$ will not actually be directly comparable since $M \parallel M'$ will contain extra outputs. Thus, we will need a way to hide these outputs.

Definition 2.14 Let M be a Moore machine and A be a set of state components. The result of *restricting M to A* (denoted $M \downarrow A$) is the Moore machine M' defined by:

1. $S' = S$.
2. $I' = I$.

3. $A'_I = A_I \cap A$.

4. $A'_O = A_O \cap A$.

5. $R'(s_0, f', s_1)$ iff there exists f such that $f' = f \downarrow A$ and $R(s_0, f, s_1)$.

6. L' is defined by $L'(s) = L(s) \downarrow A$.

While the above definition makes it possible to hide inputs, in general we will only be concerned with hiding output state components. Hiding outputs can just be thought of as "erasing" part of the output labeling on each state of the Moore machine. Now we give our definition of equivalence between Moore machines. This is essentially the standard notion of strong bisimulation [71].

Definition 2.15 Let M and M' be Moore machines with $A_I = A'_I$ and $A_O = A'_O$. $\approx \subseteq S \times S'$ is a ***bisimulation relation*** iff for every pair of states s_0 and s'_0 such that $s_0 \approx s'_0$, the following holds:

1. $L(s_0, a) = L'(s'_0, a)$ for all $a \in A_O$.

2. For all labeling functions f over A_I, if $R(s_0, f, s_1)$, there exists s'_1 such that $R'(s'_0, f, s'_1)$ and $s_1 \approx s'_1$.

3. For all labeling functions f over A_I, if $R'(s'_0, f, s'_1)$, there exists s_1 such that $R(s_0, f, s_1)$ and $s_1 \approx s'_1$.

*Two states s and s' are **bisimulation equivalent** ($M, s \equiv M', s'$) whenever there exists a bisimulation relation $\approx$ such that $s \approx s'$. M and M' are **bisimulation equivalent** ($M \equiv M'$) whenever for every state $s \in I$, there exists $s' \in I'$ such that $M, s \equiv M', s'$, and conversely, for every state $s' \in I'$, there exists $s \in I$ such that $M, s \equiv M', s'$.*

Note that if the Moore machines M and M' in the above definition are closed, then bisimulation between the Moore machines corresponds exactly to bisimulation between their structures (where structure bisimulation is defined in the standard way). Next, we turn to the proof that composing a closed Moore machine with another closing environment does not affect the first machine.

Proposition 2.1 Let M be a closed Moore machine, and M' be a closing environment for M. Then $M \equiv (M \parallel M') \downarrow A_O$.

Proof Define $M'' = M \parallel M'$ and $M''' = M'' \downarrow A_O$. Let $\approx$ be defined by $s \approx (s, s')$ for every s' in S'; we show that $\approx$ is a bisimulation relation. If $s_0 \approx (s_0, s'_0)$, then:

1. $L'''((s_0, s'_0)) = (L(s_0) \cup L'(s'_0)) \downarrow A_O = L(s_0)$.

2. Suppose $R(s_0, f, s_1)$; note that $dom(f) = \emptyset$. Since the transition relation for any Moore machine is total, there exists s'_1 such that $R'(s'_0, L(s_0) \downarrow A'_I, s'_1)$. Now by the definition of Moore machine composition, $R''((s_0, s'_0), f, (s_1, s'_1))$. This implies that (s_0, s'_0) and (s_1, s'_1) are also related via R'''. By our definition of $\approx$, $s_1 \approx (s_1, s'_1)$.

3. Suppose $R'''((s_0, s'_0), f, (s_1, s'_1))$; again note that $dom(f) = \emptyset$. Then $R''((s_0, s'_0), f, (s_1, s'_1))$. Now by the definition of Moore machine composition, $R(s_0, f, s_1)$, and we have $s_1 \approx (s_1, s'_1)$.

Moore machines must have non-empty initial state sets, so there must be some $s' \in I'$. Now if $s \in I$, then $(s, s') \in I''$ and $(s, s') \in I'''$. Also, every $(s, s') \in I'''$ is related by $\approx$ to $s \in I$. Thus $M \equiv M'''$. □

Because of the isomorphism between closed Moore machines and structures and the relation between closed Moore machine bisimulation and structure bisimulation, we find that *struct*(M) and *struct*$(M \parallel M')$ must be bisimilar. This implies that they satisfy the same CTL formulas, which is the desired result.

Next, we give the proof of theorem 2.1 (NP-hardness of the compositional model checking problem for full CTL). We will not repeat the details of the construction here (the reader may wish to look back over section 2.4).

Proof Assume that every closed system containing M satisfies the formula. Then in particular, the composition of M with an environment of the form shown in figure 2.13 must satisfy the formula. Such an environment represents a particular valuation of the variables in f. The composition of M and this environment obviously satisfies the left side of the implication, so $\bigvee_{k=0}^{m-1}$ **EF** $\neg c_k$ must also be true. This implies that some conjunct in f is false for the valuation under consideration. Since this valuation was chosen arbitrarily, we conclude that f is unsatisfiable.

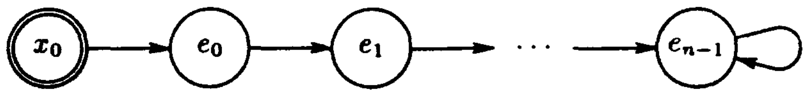


Figure 2.13: Environment representing a valuation

Assume that it is not the case that every closed system containing M satisfies the formula. Let M' be a closing environment for M for which the formula is false. Consider a run of $M \,||\, M'$. Let $i, i_0, \ldots, i_{n-1}$ be the first $n+1$ inputs supplied to M by M'. For the formula to be false,

$$\bigwedge_{j=0}^{n-1} ((\mathbf{AX}^{j+1}\, x_j) \vee (\mathbf{AX}^{j+1}\, \neg x_j))$$

must be true. Hence i_j must be either x_j or $\neg x_j$. Consider applying this sequence of inputs to the recognizer for c_k. Since

$$\bigvee_{k=0}^{m-1} \mathbf{EF}\, \neg c_k$$

must be false, this sequence of inputs must lead to the state of the recognizer labeled with c_k, i.e., c_k must be true for the valuation represented by this sequence of inputs. But since this is true for an arbitrary c_k, this valuation must in fact be a satisfying valuation for f. □

We now give the details of the approximation algorithm for solving the compositional model checking problem for CTL. Given a Moore machine M, a state s of M, and a CTL formula φ, let $M, s \models \varphi$ denote that: for every closed system containing M, every composite state in which M is at s satisfies φ. This is analogous to what it means for M to satisfy φ, but we only consider a specific state of M. Let f be a labeling function over A_I; $M, s, f \models \varphi$ will be similar to $M, s \models \varphi$, except that we only consider composite states where M is at s and the input supplied to M is f. For example, suppose M is the Moore machine of figure 2.10 and let s be the state just to the left of and below the initial state. Then $M, s, f \models \mathbf{EX}\, b = 1$ when $f(a) = 1$. The algorithm will record, for each subformula φ and each state s of M, a set of f such that $M, s, f \models \varphi$ and a set of f such that $M, s, f \models \neg\varphi$. (Since we are only computing an approximation, the sets might not include all f satisfying these conditions.)

For atomic formulas these sets are computed in the obvious way. Similarly, the sets a formula like $\varphi \wedge \psi$ can be computed in a straightforward manner from the sets for φ and ψ. The only interesting question is how to compute the sets for $\mathbf{EX}\,\varphi$ from the sets for φ. Consider applying the input f at state s. Suppose that given this input, s has successors $s_0, \ldots, s_{n-1}$. Also suppose that for one of the s_i, $M, s_i, f' \models \varphi$ for all possible f'. Then clearly $M, s, f \models \mathbf{EX}\,\varphi$. More generally, suppose that some of the s_i, say s_0 and s_1, have the same output labeling. In this case, the environment cannot distinguish between s_0 and s_1 and hence *must supply the same* inputs to both. Thus we have $M, s, f \models \mathbf{EX}\,\varphi$ if for every f', either $M, s_0, f' \models \varphi$ or $M, s_1, f' \models \varphi$. We take the union of the sets of valuations for which φ is known to be true at s_0 and s_1 and see whether this is the set of all input valuations. In summary, our strategy for deciding whether $M, s, f \models \mathbf{EX}\,\varphi$ is to look at the successors of s under f, group them into classes according to their output labeling, and take the union of the sets for which φ is true within each class. If for any class, the result is the set of all input valuations, then we know $M, s, f \models \varphi$. (Note we are actually doing some work to try to resolve situations involving nondeterministic transitions. However, we are bounding the amount of "lookahead" that we are willing to do to just one level of successors.) Now consider $\neg\,\mathbf{EX}\,\varphi$. This formula must be true at s if for every successor s_i, φ is known to be false at s_i for every input valuation, i.e., $M, s_i, f' \models \neg\varphi$ for all s_i and f'.

We will let $S_{s,\varphi}$ denote the set of input valuations f for which we know that $M, s, f \models \varphi$. Also, $T_{s,\varphi}$ will denote the set of input valuations f for which we know $M, s, f \models \neg\varphi$. In figures 2.14 through 2.16, we give the algorithm for computing $S_{s,\varphi}$ and $T_{s,\varphi}$ for all states s and subformulas φ. We have omitted the description of the procedure *computeegsets* since it is similar to *computeeusets* except using the fixed point characterization of $\mathbf{EG}$. Also, all the assignments of the form $S_{s,\varphi} := \ldots$ and $T_{s,\varphi} := \ldots$ include an implicit loop over all states s of M.

```
procedure computesets($\varphi$)
if $\varphi = (a = d)$
    $S_{s,\varphi} := \{ f \mid (L(s) \cup f)(a) = d \}$
    $T_{s,\varphi} := \{ f \mid (L(s) \cup f)(a) \neq d \}$
else if $\varphi = (\neg\psi)$
    computesets($\psi$)
    $S_{s,\varphi} := T_{s,\psi}$
    $T_{s,\varphi} := S_{s,\psi}$
else if $\varphi = (\psi_0 \wedge \psi_1)$
    computesets($\psi_0$)
    computesets($\psi_1$)
    $S_{s,\varphi} := S_{s,\psi_0} \cap S_{s,\psi_1}$
    $T_{s,\varphi} := T_{s,\psi_0} \cup T_{s,\psi_1}$
else if $\varphi = (\mathbf{EX}\,\psi)$
    computeexsets($\psi$)
else if $\varphi = (\mathbf{E}(\psi_0\,\mathbf{U}\,\psi_1))$
    computeeusets($\psi_0, \psi_1$)
else if $\varphi = (\mathbf{EG}\,\psi)$
    computeegsets($\psi$)
endif
```

Figure 2.14: Approximation algorithm for the compositional model checking problem for CTL

To show correctness, we have the following.

Proposition 2.2 For all subformulas φ and all states s and input valuations f, we have:

1. if $f \in S_{s,\varphi}$, then $M, s, f \models \varphi$; and

2. if $f \in T_{s,\varphi}$, then $M, s, f \models \neg\varphi$.

```
function $ex(s, \varphi)$
$result := \emptyset$
for each input valuation $f$
    for each class $C$ of successors of $s$ under $f$
    with identical output labelings
        if $\bigcup_{s' \in C} S_{s',\varphi}$ includes all input labeling functions
            $result := result \cup \{f\}$
        endif
    endfor
endfor
return $result$

function $ax(s, \varphi)$
$result := \emptyset$
for each input valuation $f$
    if for every successor $s'$ of $s$ under $f$,
    $T_{s',\varphi}$ includes all input labeling functions
        $result := result \cup \{f\}$
    endif
endfor
return $result$

procedure $computeexsets(\varphi)$
$computesets(\varphi)$
$S_{s,\mathbf{EX}\,\varphi} := ex(s, \varphi)$
$T_{s,\mathbf{EX}\,\varphi} := ax(s, \varphi)$
```

Figure 2.15: Procedure for computing S and T for $\mathbf{EX}\,\varphi$

```
procedure $computeeusets(\varphi, \psi)$
$computesets(\varphi)$
$computesets(\psi)$
$S_{s,Y} := \emptyset$
repeat
    $S_{s,Y} := S_{s,\psi} \cup (S_{s,\varphi} \cap ex(s, Y))$
until fixed point
$S_{s,\mathbf{E}(\varphi\mathbf{U}\psi)} := S_{s,Y}$
set $T_{s,Y}$ to the set of all input valuations
repeat
    $T_{s,Y} := T_{s,\psi} \cap (T_{s,\varphi} \cup ax(s, Y))$
until fixed point
$T_{s,\mathbf{E}(\varphi\mathbf{U}\psi)} := T_{s,Y}$
```

Figure 2.16: Procedure for computing S and T for $\mathbf{E}(\varphi\ \mathbf{U}\ \psi)$

Proof By induction on the structure of the subformula. For atomic subformulas, the result is trivial, and for subformulas whose top operator is a logical connective, the result follows in a straightforward way from the induction hypothesis.

Consider a subformula $\mathbf{EX}\,\varphi$; suppose $f \in S_{s,\mathbf{EX}\,\varphi}$. Then there exists a group $s_0, \ldots, s_{n-1}$ of successors of s under f such that $L(s_i) = L(s_j)$ for all i and j and such that $\bigcup_i S_{s_i,\varphi}$ is the universal set of input labeling functions. Consider any closing environment that presents f to M at the state s. M will be able to make transitions to all of the s_i, and if the environment presents an input f' to one s_i, it must present that same input to all. But for any such f', there exists an i such that $f' \in S_{s_i,\varphi}$. By the induction hypothesis, $M, s_i, f' \models \varphi$, so in the environment that we are considering, φ will be satisfied starting at s_i. Hence, given the input f, s will have a successor satisfying φ, i.e., $M, s, f \models \mathbf{EX}\,\varphi$.

Suppose now that $f \in T_{s,\mathbf{EX}\,\varphi}$. For every successor s' of s under f, $T_{s',\varphi}$ is the set of all input valuations. By the induction hypothesis, $M, s', f' \models \neg\varphi$ for every f'. Thus, given the input f, φ must be false regardless of the closing environment. Hence $M, s, f \models \neg\,\mathbf{EX}\,\varphi$.

Next, we consider subformulas of the form $\mathbf{E}(\varphi \mathbf{U} \psi)$. Assume that $f \in S_{s,\mathbf{E}(\varphi\mathbf{U}\psi)}$, and fix a closing environment for M that supplies f at s. Since $f \in S_{s,\mathbf{E}(\varphi\mathbf{U}\psi)}$, there is some iterate of $S_{s,Y}$, say $S_{s,Y,i}$, containing f. Assume without loss of generality that i is chosen to be as small as possible. We prove by induction on i that $M, s, f \models \mathbf{E}(\varphi \mathbf{U} \psi)$. If $i = 1$, then we must have $f \in S_{s,\psi}$. By the outer induction hypothesis, $M, s, f \models \psi$, and hence $M, s, f \models \mathbf{E}(\varphi \mathbf{U} \psi)$. For $i > 1$, we have $f \in S_{s,\varphi}$ (so $M, s, f \models \varphi$). Further, given f, s must have a class of successors $s_0, \ldots, s_{n-1}$ such that all s_j have the same labeling and for each f', $f' \in S_{s_j,Y,i-1}$ for some j. By the inner induction hypothesis, $M, s_j, f' \models \mathbf{E}(\varphi \mathbf{U} \psi)$. This implies $M, s, f \models \mathbf{EX}\,\mathbf{E}(\varphi \mathbf{U} \psi)$. As a result, $M, s, f \models \mathbf{E}(\varphi \mathbf{U} \psi)$.

Suppose $f \in T_{s,\mathbf{E}(\varphi\mathbf{U}\psi)}$, and again fix a closing environment that supplies f at s. Consider the iterates $T_{s,Y,i}$ for $i > 0$. We prove via induction on i that if $f \in T_{s,Y,i}$ then there is no path starting at s and beginning with the input f satisfying $\varphi \mathbf{U} \psi$ and such that a state satisfying ψ is reached within $i - 1$ steps. For $i = 1$, we have $f \in T_{s,\psi}$, and so by the outer induction hypothesis, $M, s, f \models \neg\psi$. For $i > 1$, assume we have a path satisfying $\varphi \mathbf{U} \psi$. We know $f \in T_{s,\psi}$, so again ψ cannot be true immediately. Thus, since the path satisfies $\varphi \mathbf{U} \psi$, it must satisfy φ at s. This implies $f \notin T_{s,\varphi}$, so for every successor of s under f, and for the successor s' on the path in particular, $T_{s',Y,i-1}$ is the universal set of input valuations. Now by the induction hypothesis, there is no path starting at s' satisfying $\varphi \mathbf{U} \psi$ and such that a state satisfying ψ is reached within $i - 2$ steps. Hence, ψ is not reached in $i - 1$ steps on the original path. Now suppose that there is in fact a path satisfying $\varphi \mathbf{U} \psi$ from s and beginning with the input f. Since ψ must become true at some point on this path, f must not be in $T_{s,Y,i}$ for sufficiently large i. But this implies $f \notin T_{s,\mathbf{E}(\varphi\mathbf{U}\psi)}$, a contraction. Hence there is no such path, and so $M, s, f \models \neg\,\mathbf{E}(\varphi \mathbf{U} \psi)$.

The proof for subformulas of the form $\mathbf{EG}\,\varphi$ is similar in spirit to the above and is omitted. □

Our final proof is of theorem 2.2 (that it is enough to check ACTL formulas just using the maximal closing environment.)

Proof Let M' be a closing environment for M. Then we know that $A'_O \supseteq A_I$. This implies that for state s' of M', we can derive a unique labeling function over A_I by $L'(s') \downarrow A_I$. Now each such labeling function is a state of $E(M)$, so the map ϕ defined by $\phi(s') = L'(s') \downarrow A_I$ maps states of M' to states of M. We can extend this to a map from states of $M \parallel M'$ to states of $M \parallel E(M)$ by having $\phi((s, s')) = (s, L'(s') \downarrow A_I)$. Now:

1. If (s, s') is an initial state of $M \parallel M'$, then $s \in I$, and so $\phi((s, s'))$ is an initial state of $M \parallel E(M)$.

2. If (s_0, s'_0) can make a transition to (s_1, s'_1) in $M \parallel M'$ (remember there are no free inputs), then $R(s_0, L'(s'_0) \downarrow A_I, s_1)$. This implies that $\phi((s_0, s'_0))$ can make a transition to $\phi((s_1, s'_1))$ in $M \parallel E(M)$. Hence, every path in $M \parallel M'$ has a corresponding path in $M \parallel E(M)$.

Since Moore machines with no free inputs are isomorphic to their corresponding structures, we will ignore the distinction for the remainder of this proof.

We now prove by induction on the structure of ACTL formulas that if $M \parallel E(M), \phi((s, s')) \models \varphi$, then $M \parallel M', (s, s') \models \varphi$.

1. For *true*, we trivially have $\phi((s, s')) \models$ *true* and $(s, s') \models$ *true*.

2. Consider the atomic formula $a = d$. Assume $a \in A_O$; then $\phi((s, s')) = (s, L'(s') \downarrow A_I)$, so $\phi((s, s')) \models (a = d)$ iff $L(s, a) = d$. However, $(s, s') \models (a = d)$ iff $L(s, a) = d$ as well. If $a \in A_I$, then $\phi((s, s')) \models (a = d)$ iff $(L'(s') \downarrow A_I)(a) = d$ iff $L'(s', a) = d$ iff $(s, s') \models (a = d)$.

3. For negations of atomic formulas, just note that in the above two cases, we showed iff's rather than simple implication.

4. For conjunctions and disjunctions, the result follows immediately from the induction hypothesis.

5. Consider a formula of the formula $\mathbf{A}(\varphi \mathbf{U} \psi)$. Assume $\phi((s, s'))$ satisfies this formula. Let $(s_0, s'_0)(s_1, s'_1) \ldots$ be a path in $M \parallel M'$ from $(s, s') = (s_0, s'_0)$. Assume that this path does not satisfy $\varphi \mathbf{U} \psi$. Then there exists j such that (s_j, s'_j) does not satisfy φ,

and for all $i \leq j$, (s_i, s'_i) does not satisfy ψ. By the (contrapositive of the) induction hypothesis, $\phi((s_j, s'_j)) \not\models \varphi$ and $\phi((s_i, s'_i)) \not\models \psi$ for all $i \leq j$. However, $\phi((s_0, s'_0))\phi((s_1, s'_1))\ldots$ is a path in $M \parallel E(M)$. This path does not satisfy $\varphi \mathbf{U} \psi$, and so $\phi((s_0, s'_0)) = \phi((s, s'))$ does not satisfy $\mathbf{A}(\varphi \mathbf{U} \psi)$, a contradiction. Thus the path $(s_0, s'_0)(s_1, s'_1)\ldots$ must in fact satisfy $\varphi \mathbf{U} \psi$. However, this path was chosen arbitrarily, and so $(s, s') \models \mathbf{A}(\varphi \mathbf{U} \psi)$. The proof for the other temporal operators is similar.

Now we have shown that $\phi((s, s')) \models \varphi$ implies that $(s, s') \models \varphi$. If $M \parallel M' \not\models \varphi$, then there is an initial state (s, s') of $M \parallel M'$ such that $(s, s') \not\models \varphi$. By the above, $\phi((s, s')) \not\models \varphi$. But $\phi((s, s'))$ is also an initial state, and so $M \parallel E(M) \not\models \varphi$. Taking the contrapositive gives that $M \parallel E(M) \models \varphi$ implies $M \parallel M' \models \varphi$. Since M' was an arbitrary closing environment, we conclude that if $M \parallel E(M) \models \varphi$, then $M \models \varphi$. Finally, if $M \parallel E(M) \not\models \varphi$, then there is a closed system containing M that does not satisfy φ, and so $M \not\models \varphi$. □

Chapter 3

Compositional Verification, Part II

In the previous chapter, we considered the problem of determining whether a temporal logic formula φ is true of all closed systems that can be built using a component M. In practice however, we need more powerful capabilities:

1. We need to be able to do *assume-guarantee* style reasoning. Components are generally designed with some assumptions about how their environment will behave. Thus, we want to check that: for all closing environments, either the environment violates some assumption, or the composition of M with the environment is guaranteed to satisfy φ.

2. We need methods of doing *hierarchical verification*. In hierarchical verification, the specifications that we check become implementations at the next higher level of abstraction. When our specifications are given as formulas and our components are given as state transition systems, it is not obvious how this can be achieved.

Consider, for example, a pair of components M and M' that work together to provide a service to a larger environment. The environment passes requests to M, and M enqueues them. M' removes requests from the queue, processes them, and sends acknowledgments back to the environment. Suppose that we wish to verify that every request that the environment makes is eventually acknowledged. We may try to deduce this by verifying that:

1. every request that M receives is eventually enqueued; and

2. every request that is put on the queue is eventually processed and acknowledged by M'.

The first property above is essentially a local property of M, while the second is a local property of M'. Thus, we might try to check the properties using just M and just M', respectively. However, if M and M' have been designed with some assumptions about the protocol used to access the queue, then we may find that the "local" properties really depend on these assumptions. When doing the verification, we must take these assumptions into account. (Of course, we must also discharge the assumptions by showing that M and M' follow the intended protocol.) Suppose that we do manage to verify that every request made by the environment is eventually acknowledged, and that we now want to prove a global progress property about the whole system. This progress property may depend on the fact that M and M' eventually service requests. However, it probably does not depend on the details of how this is accomplished. Thus, instead of using M and M' when doing the verification, we would like to use the first property that we checked as an alternative "implementation" to $M \parallel M'$.

In this chapter, we show how to do assume-guarantee style reasoning and hierarchical verification using ACTL. This is achieved by proving a correspondence between satisfaction of ACTL formulas and a type of *simulation relation* between structures. We also illustrate these ideas by verifying the controller for a simple stack-based CPU.

3.1 Assume-Guarantee Reasoning

The *assume-guarantee* style of verification was first advocated in the context of temporal logic by Pnueli [77]. In Pnueli's system, we work with triples of the form $\langle\varphi\rangle M\langle\psi\rangle$. The most common reading of such a triple is "if the environment of M satisfies φ, then M in this environment satisfies ψ." A typical chain of reasoning would be as follows:

$$\frac{\begin{array}{c}\langle\rangle M\langle\varphi\rangle\\ \langle\varphi\rangle M'\langle\psi\rangle\end{array}}{\langle\rangle M \parallel M'\langle\psi\rangle.}$$

Here, we are asserting that if:

1. M satisfies φ; and

2. if the environment of M' satisfies φ, then M' satisfies ψ

then the composition of M and M' will satisfy ψ. The advantage of doing the verification in this manner is that we never have to examine the composite state space of $M \parallel M'$. Instead, we check φ using just M, and then check ψ using only M' and the (hopefully simple) assumption φ. The disadvantage is that the user must determine an appropriate φ. As we shall demonstrate later however, knowledge of how the system should behave plus feedback from an automatic verifier makes this feasible in practice.

More generally, we may use multiple levels of assumptions and guarantees when doing a verification. That is, once we have proved a guarantee, we may use that guarantee as an assumption in later stages. Because of this, a somewhat more precise reading of $\langle\varphi\rangle M\langle\psi\rangle$ would be "if the system satisfies φ and contains M, then the system also satisfies ψ." This is because φ may in fact be something that is derived based on earlier assumptions about M, and may reflect these assumptions. Also, ψ may describe the combination of M and its environment, instead of just M. Of course, in order to avoid erroneous conclusions, all chains of deduction must be well-founded, i.e., the base assumptions must themselves be proved without any assumptions. There is a natural temptation to argue that $\langle\varphi\rangle M\langle\psi\rangle$ and $\langle\psi\rangle M'\langle\varphi\rangle$ should be sufficient to conclude $\langle\rangle M \parallel M'\langle\varphi\rangle$ and $\langle\rangle M \parallel M'\langle\psi\rangle$, but such circular reasoning is generally not sound.

Example 3.1 Consider the Moore machine for the circuit of figure 2.2. For convenience, the Moore machine is reproduced in figure 3.1; we will call it M. Assume that we wish to prove that the composition of M

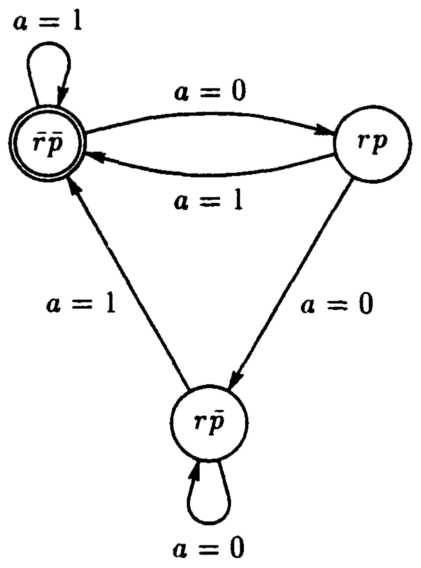


Figure 3.1: Moore machine for the circuit of figure 2.2

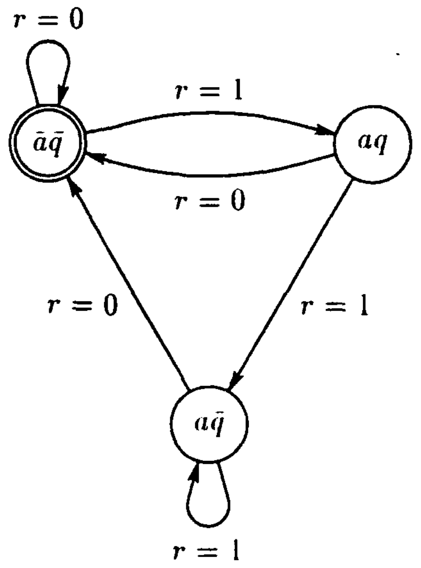


Figure 3.2: Moore machine for the circuit of figure 2.4

with the Moore machine M' of figure 3.2 (representing the circuit of figure 2.4) satisfies the specification $\mathbf{AG}(p = 0 \vee q = 0)$. We can see that this should be true since:

1. M only sets $p = 1$ at the same instant that it first sets $r = 1$; and
2. M' sets $q = 0$ when it observes $r = 0$, and does not set $q = 1$ until one step after it observes $r = 1$.

So, when r first changes from 0 to 1, p does so simultaneously. At that point, q is still 0, since the change in r has not been observed yet. One step later, p changes back to 0, while q observes the change in r and transitions to 1.

We can verify the specification using assume-guarantee style reasoning as follows. First, we express the above assumption about M': $\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)$. Next, we check that M' in fact satisfies this assumption. Finally we use this assumption to show that M satisfies the desired specification, and conclude that $M \parallel M'$ satisfies the specification.

$$\frac{\begin{array}{c}\langle\rangle M' \langle \mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)\rangle \\ \langle \mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)\rangle M \langle \mathbf{AG}(p = 0 \vee q = 0)\rangle\end{array}}{\langle\rangle M \parallel M' \langle \mathbf{AG}(p = 0 \vee q = 0)\rangle}$$

In the next section, we will consider how we actually go about establishing the truth of a triple $\langle\varphi\rangle M \langle\psi\rangle$. □

3.2 Framework

In this section, we describe the basic framework for supporting assume-guarantee style reasoning. (We presented this framework in 1991 [52]). To provide a unified basis for doing assume-guarantee style reasoning and compositional verification, we are going to introduce a notion of *simulation* between state transition systems. Intuitively, the simulation relation $\preceq$ will capture the notion of what it means for one such system to include "more behaviors" than another. This notion is in fact implicit in the section on ACTL in the previous chapter. There we showed that checking $M \models \varphi$, where M is a Moore machine, could be done by composing M with its maximal environment. In a sense, the maximal environment, which can provide any input at any point, has more behaviors than any other environment. Put another way, the maximal environment can simulate any other environment; our proof of theorem 2.2 was based on this idea.

The relation $\preceq$ will be a preorder, i.e., a reflexive and transitive relation. We could in fact view simulation as the basic relationship between an implementation and a specification. Because of the transitivity of $\preceq$, we would get hierarchical verification essentially for free. For example, if $M \preceq M'$ ("M' can simulate M"), and if we want to know whether $M \preceq M''$, then it would be enough to check that $M' \preceq M''$. Here, M' would represent a specification of M that is used to prove a higher level specification M''. The simulation relation will also interact with composition in a nice way: if $M \preceq M'$, then we will have $M \parallel M'' \preceq M' \parallel M''$. This type of property allows us to replace an implementation by its specification in a composition. It also gives us the analog of theorem 2.2; if we want to check $M \parallel M' \preceq M''$, where M'' is conceptually a local property of M, then we can use a maximal environment $E(M)$ for M. That is, we will have $M' \preceq E(M)$, and so $M \parallel M' \preceq M \parallel E(M)$. Then by checking $M \parallel E(M) \preceq M''$, we can use transitivity to conclude $M \parallel M' \preceq M''$.

Previously, we had one notion of satisfaction of a temporal logic specification ($\models$). Above, we have suggested a notion of satisfaction of an automata specification ($\preceq$). We would like to have some correspondence between these two notions. This is done via a *tableau construction* that maps a formula φ to an associated state transition system $\mathcal{T}(\varphi)$ which is called the tableau of the formula. We will prove that satisfaction of a formula corresponds exactly to being simulated by the tableau for the formula. Thus, $\preceq$ and $\models$ will really turn out to be compatible notions. Further, the tableau construction makes it clear how to do hierarchical reasoning with specifications that are given as formulas. We simply use the standard model checking algorithm at one level, then construct tableaus for the specification formulas and use them as implementations at the next higher level. The tableau construction can also be used for doing things like checking implication between temporal formulas. Viewed another way, the correspondence between state transition systems and formulas allows us to mix and match, using either formulas or automata as either implementations or specifications, whichever is most convenient.

Finally, with the above framework, it is easy to do assume-guarantee style reasoning. The key observation is that an assumption (specified, e.g., as a formula φ) represents the maximal environment that satisfies that assumption. Consider the following assume-guarantee proof:

$$\frac{\begin{array}{c}\langle\rangle M\langle\varphi\rangle\\ \langle\varphi\rangle M'\langle\psi\rangle\end{array}}{\langle\rangle M \parallel M'\langle\psi\rangle.}$$

We interpret $\langle\rangle M\langle\varphi\rangle$ as saying that all the behaviors of M can be simulated by $\mathcal{T}(\varphi)$, i.e., $M \preceq \mathcal{T}(\varphi)$. Because of the correspondence between $\preceq$ and $\models$, this can be checked by verifying $M \models \varphi$. Eventually, we want to conclude that $M \parallel M' \preceq \mathcal{T}(\psi)$. To check $\langle\varphi\rangle M'\langle\psi\rangle$, we use $\mathcal{T}(\varphi)$ as the maximal environment satisfying φ, i.e., we verify $\mathcal{T}(\varphi) \parallel M' \models \psi$. This is equivalent to saying $\mathcal{T}(\varphi) \parallel M' \preceq \mathcal{T}(\psi)$. Since $M \preceq \mathcal{T}(\varphi)$, we can compose both sides with M' to obtain $M \parallel M' \preceq \mathcal{T}(\varphi) \parallel M'$. Then by transitivity, $M \parallel M' \preceq \mathcal{T}(\psi)$.

As a final note, we will actually be working with structures rather than Moore machines. This is mainly because formulas do not have notions of inputs and outputs, so the tableau construction will most naturally produce structures. In addition, structures can serve as a kind of "intermediate language" for representing other, more complex types of models, such as Mealy machines [69].

3.3 Structures

In this section, we are concerned with two things. First, we are going to extend the definition of structure to include a kind of *infinitary acceptance condition.* Such an acceptance condition is used to rule out certain infinite paths through the structure. This is necessary in order to be able to define tableaus for all of the formulas in ACTL, and also to be able to make accurate models of real components. Second, since we are going to be working with structures, we need to define a notion of composition. When the structures represent Moore machines, the definition will correspond to Moore machine composition.

To see why the current notion of structure is inadequate for representing tableaus for all ACTL formulas, we look at a specific example.

Example 3.2 Suppose that a is a state component ranging over $\{0, 1\}$, and consider the formula $\mathbf{AF}\, a = 1$. Intuitively, the tableau is going to represent all those behaviors that are consistent with the formula. Thus, a first guess at the tableau might be the structure shown in figure 3.3. The idea is that starting from one of the initial states, we

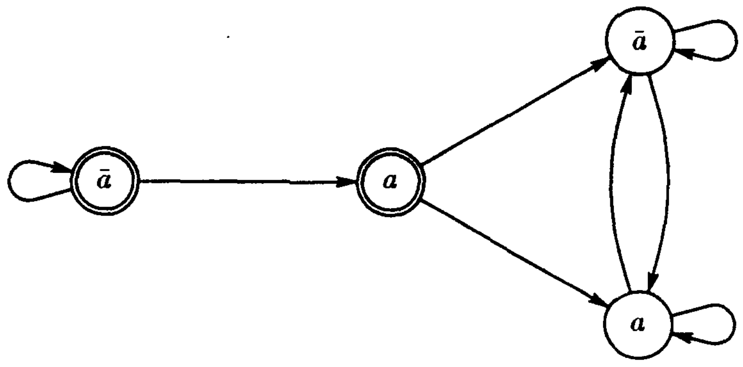


Figure 3.3: Proposed tableau for $\mathbf{AF}\, a = 1$

should eventually reach the initial state with $a = 1$. At that point, we know that the requirement that a eventually become 1 has been fulfilled. The transitions from then on are completely unconstrained. The problem of course is that there is nothing to guarantee that the initial state where $a = 1$ is eventually reached. In particular, the structure of figure 3.3 allows the behavior where a remains 0 forever. Thus, this structure would be able to simulate a structure with one (initial) state s_0 where $L(s_0, a) = 0$ and the only transition is from s_0 to s_0. Since the latter structure obviously should not satisfy $\mathbf{AF}\, a = 1$, we cannot use the structure of figure 3.3 as the tableau for $\mathbf{AF}\, a = 1$. □

In order to avoid this problem, we add another element F to structures. F will represent an infinitary acceptance condition, as used in automata on infinite strings. There are a number of different types of acceptance conditions. One that we will sometimes use for explanatory purposes is ***Büchi acceptance*** [20]. In the case of Büchi acceptance, F is a set of states. A path within the structure will be considered legal if there is some state in F that occurs an infinite number of times on the path.

Example 3.3 Consider our previous attempt to construct a tableau for $\mathbf{AF}\, a = 1$. Suppose that we let F be the set consisting of the two non-initial states of the structure of figure 3.3. Now the execution in which we continually loop in the initial state where $a = 0$ is not legal, because it does not visit any of the states in F an infinite number of times. (In fact, it never visits any of the states in F at all.) □

Büchi acceptance is sufficient to define tableaus for all ACTL formulas. We will also use infinitary acceptance conditions in making models of components. This is done for two reasons:

1. When hiding internal details of components or modeling classes of components, we use acceptance conditions to capture the notion of "arbitrary but finite" delays.

2. Some components are nondeterministic, but have probabilistic guarantees of *fairness*.

Example 3.4 We consider the example of a countdown timer. A countdown timer has an input r (for "reset") and an output e ("expired"). When r becomes 1, an internal counter is reset to some fixed starting value; also, the e output is set to 0. After r becomes 0, the internal counter starts to decrement, and when the counter reaches 0, it halts and e becomes 1. Then e remains at 1 until the next reset. Figure 3.4 shows a Moore machine for a countdown timer with a countdown value of 3. Suppose that we are verifying a system containing such a timer, and that the property we are checking does not depend on the exact number of steps that the timer takes to reach 0. In this case, we can eliminate some of the internal state of the timer model in order to try to simplify the verification. We will use abstract model of the timer shown in figure 3.5. Now we want to ensure that if r becomes 0 and remains 0, then eventually e must change to 1. This may be done by adding the acceptance condition defined by $\mathbf{GF}(r = 0 \rightarrow e = 1)$. Here,

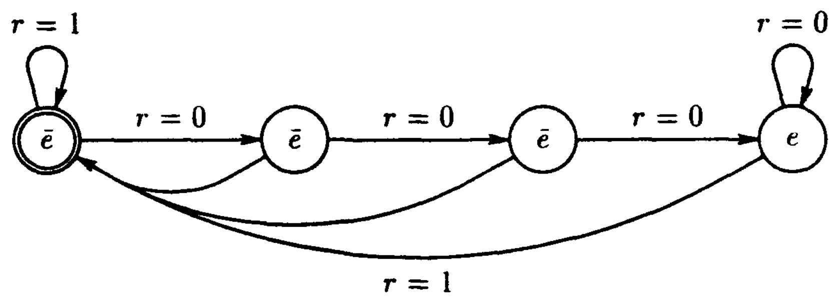
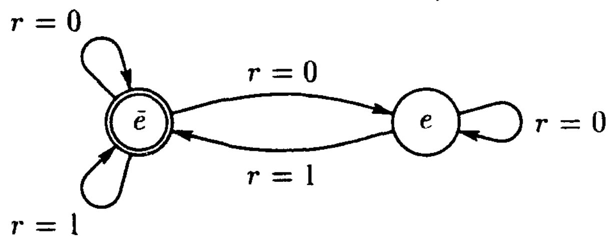


Figure 3.4: Model of a specific countdown timer

GF is a temporal operator indicating "infinitely often". The intention is that we expand out the Moore machine shown in the figure into its corresponding structure, and those states for which $r = 1$ or $e = 1$ become the elements of F. Using the abstract model has another benefit aside from simplifying the verification. In particular, if we were to change the design by substituting a different countdown timer, we would not have to re-verify those properties that we checked using the abstract model. □



$$\mathbf{GF}(r = 0 \rightarrow e = 1)$$

Figure 3.5: Abstract model of a countdown timer

Example 3.5 Suppose that we are modeling an arbiter. An arbiter is a device that receives requests from a number of agents and grants them mutually exclusive access to a shared resource. In addition to making sure that the arbiter only grants the resource to one agent at a time, we may want to say that the arbiter is fair, i.e., it should not ignore a request from any agent indefinitely. Suppose that the arbiter is in a state where *deciding* = 1 when it is about to grant the resource to an agent. Also, assume that agent i makes its request by setting the input r_i to 1 and is granted the resource when the arbiter sets the output a_i to 1. When the agent finishes using the resource, it sets r_i to 0, after which the arbiter sets a_i to 0 and goes back to the *deciding* state. Our first temptation is to say that $r_i \rightarrow a_i$ should be true infinitely often (for each i). The idea would be that when r_i is 1 and a_i is 0, the arbiter is ignoring the agent, and it should not be allowed to do so forever. Suppose, however, that agent 0 makes a request and is granted the resource, and then never releases it. Now if agent 1 makes a request, obviously it cannot be allowed to have the resource until agent 0 releases it. Further, there is no way to compel agent 0 to do so. Thus, the execution where agent 0 hogs the resource should be legal, but it is disallowed by the constraint that $r_1 \rightarrow a_1$ be true infinitely often. In short, by trying to state that the arbiter is fair to agent 1, we have restricted the legal input sequences for agent 0. This is obviously not acceptable in an accurate model of the arbiter. The real constraint that we want to specify is "if infinitely often the arbiter has a chance to make a decision and agent i is requesting the resource, then infinitely often agent i should be granted the resource". This should be true for every i:

$$\bigwedge_i (\mathbf{GF}(\mathit{deciding} \wedge r_i) \rightarrow \mathbf{GF}\, a_i).$$

This type of constraint cannot be captured using simple Büchi acceptance conditions. That is, Büchi acceptance conditions are generally not powerful enough to be able to make accurate models when doing compositional reasoning. Thus, we will actually use a stronger form of acceptance condition called *Streett acceptance* [87]. Streett acceptance can express constraints like the one above. F will be a set of pairs (P, Q) of sets of states. A path is legal if for every (P, Q), either the path stays inside P after some point, or infinitely often it visits a state in Q. That is

$$\bigwedge_{(P,Q)} (\mathbf{FG}\, P \vee \mathbf{GF}\, Q).$$

FG is a temporal operator expressing "almost always": it is the dual of **GF**. (In the context of ω-regular language theory, the two types of acceptance conditions are equivalent, provided the automata are allowed to be nondeterministic. However, if we try to change our arbiter example to use Büchi acceptance by adding nondeterminism, we have to alter the branching structure. Since we are working in a branching-time framework, this is not acceptable.) □

We now give the extended definition of a structure that includes an infinitary acceptance condition. Previous constructions involving structures will be extended in the obvious way. For example, when constructing the structure for a Moore machine, we construct S, I, etc., as before and take $F = \emptyset$.

Definition 3.1 A *structure* $M = \langle S, I, R, A, L, F\rangle$ is a tuple of the following form:

1. *S, I, R, A, and L are as in definition 2.2.*
2. F is a set of pairs of subsets of S.

We also add the requirement that a sequence of states which is to be considered a path must fulfill the acceptance condition. This extends to the semantics of CTL and ACTL: the **A** and **E** quantifiers will range only over such sequences.

Definition 3.2 Assume M is a structure, and let $\pi = s_0 s_1 s_2 \ldots$ be a sequence of states of M. We define $inf(\pi)$ to be those s_i such that s_i appears infinitely often in π. We say that π is a *path* in M starting at s_0 when:

1. for all i, $R(s_i, s_{i+1})$; and
2. for every $(P, Q) \in F$, either $inf(\pi) \subseteq P$ or $inf(\pi) \cap Q \neq \emptyset$.

We now turn to the definition of composition of structures. As mentioned before, we want this definition to correspond to Moore machine composition in the case that the structures represent Moore machines. That is, we want the following property to hold:

Proposition 3.1 Let M and M' be Moore machines that can be composed. Then *struct*$(M \parallel M')$ is isomorphic to *struct*$(M) \parallel$ *struct*(M').

With this in mind, we consider a specific example to motivate the definition.

Example 3.6 Recall the request-acknowledge circuits that we used as examples in chapter 2. The structures for the Moore machines representing the circuits of figures 2.2 and 2.4 are reproduced in figures 3.6 and 3.7. We call these structures M and M', respectively. The

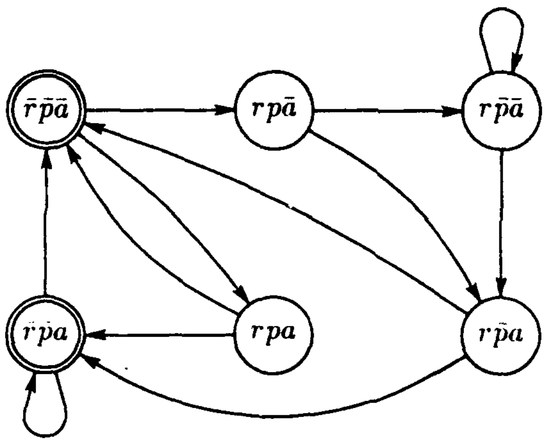


Figure 3.6: Structure for the Moore machine shown in figure 3.1

structure representing the composite Moore machine is shown in figure 3.8 (this is the reachable portion of the state space only). Consider the state $rp\bar a\bar q$ in the composition. When we project this down onto the sets of state components $A = \{r, p, a\}$ and $A' = \{r, a, q\}$, we obtain labelings $rp\bar a$ and $\bar a\bar qr$. Thus, it seems natural to view the state $rp\bar a\bar q$ as being represented by a pair of states, $rp\bar a$ in M and $\bar a\bar qr$ in M'. Since Moore machine composition is synchronous, composition of structures should be as well, i.e., in a step of the composition, both parts should make transitions. The successors of $rp\bar a$ are $r\bar p\bar a$ and $r\bar pa$, and the successors of $\bar a\bar qr$ are aqr and aqr. If we look at pairs of these successors, only $r\bar pa$ and aqr have "compatible" labelings. Now the only successor

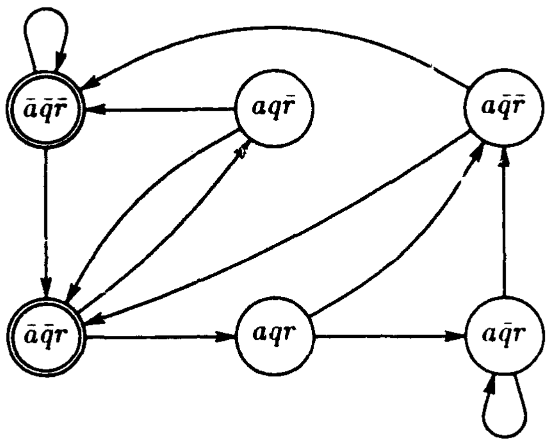


Figure 3.7: Structure for the Moore machine shown in figure 2.6

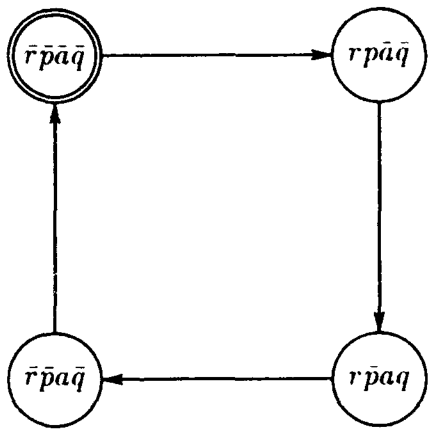


Figure 3.8: Structure representing the composite Moore machine

of $r\bar{p}\bar{a}\bar{q}$ in the composition is $r\bar{p}aq$, which in fact does project down to this pair. What about the other pairs though? If we turn back to our physical model of circuits, we see that a pair such as $r\bar{p}\bar{a}$ and aqr represents a situation in which one part of the circuit sees the logic value 0 on the wire a, and the other sees the logic value 1 on the same wire. Since this violates our physical intuition, we shall simply eliminate pairs of states with incompatible labelings from the composition. As for the initial states of the composition, we note that $\bar{r}\bar{p}\bar{a}\bar{q}$, which is an initial state, projects to initial states in both M and M'. On the other hand, a state such as $rpa\bar{q}$, which projects to an initial state in M' but not in M, should not be initial. In summary, to obtain the relationship of proposition 3.1, we should view states of the composition as pairs of component states with compatible labelings. Transitions should correspond to transitions in each component structure, and initial states should correspond to pairs of initial states. Under this interpretation, the composition of M and M' will in fact give rise to the structure in figure 3.8. □

The only minor issue that remains is how to define the acceptance conditions for a composition. We will do it in such a way that a path in the composition corresponds to paths in the components. Also, given a pair of paths in the components such that the labelings along the two paths are compatible, we should be able to lift the pair to a path of the composition. Consider a sequence of states $(s_0, s'_0)(s_1, s'_1)\ldots$ in the composition of M and M'. In order to ensure that $s_0 s_1 \ldots$ represents a path in the first component, we want to check that for each $(P, Q) \in F$, either the s_i are eventually entirely within P or infinitely often visit Q. This is equivalent to the (s_i, s'_i) eventually being entirely within $P \times S'$ or infinitely often visiting $Q \times S'$. Each acceptance condition pair in F and F' is lifted to a pair for the composition in this way.

Definition 3.3 Let M and M' be two structures. The *composition of M and M'*, denoted $M \parallel M'$, is the structure M'' defined as follows:

1. S'' is the set of pairs $(s, s') \in S \times S'$ for which $L(s, a) = L'(s', a)$ for all a in $A \cap A'$.

2. $I'' = (I \times I') \cap S''$.

3. $R''((s_0, s'_0), (s_1, s'_1))$ iff $R(s_0, s_1)$ and $R'(s'_0, s'_1)$.

4. $A'' = A \cup A'$.

5. $L''((s, s'), a) = L(s, a)$ for all a in A. $L''((s, s'), a') = L'(s', a')$ for all a' in A'.

6. $$F'' = \{ ((P \times S') \cap S'', (Q \times S') \cap S'') \mid (P, Q) \in F \}$$ $$\cup \{ ((S \times P') \cap S'', (S \times Q') \cap S'') \mid (P', Q') \in F' \}.$$

At the end of this chapter, we will give proofs that the above definition of composition is commutative and associative (up to isomorphism), and also that proposition 3.1 holds.

3.4 Simulation Relations

Now we proceed to the definition of simulation. The intuition is similar to that behind traditional Milner-style simulation [70], except that we consider infinite paths instead of single transitions.

Definition 3.4 Let M and M' be two structures with $A \supseteq A'$. A relation $\sqsubseteq$ over $S \times S'$ is a *simulation relation* between M and M' if for all s and s' satisfying $s \sqsubseteq s'$, the following conditions hold:

1. $L(s, a') = L'(s', a')$ for all a' in A'.

2. For every path $\pi = s_0 s_1 s_2 \ldots$ starting at $s = s_0$, there exists a path $\pi' = s'_0 s'_1 s'_2 \ldots$ starting at $s' = s'_0$ such that for all i, $s_i \sqsubseteq s'_i$.

The *state s of M is simulated by the state s' of M'* ($M, s \preceq M', s'$) whenever there exists a simulation relation $\sqsubseteq$ between M and M' such that $s \sqsubseteq s'$. (We often omit M and M' when they are clear from context.) *M' simulates M* ($M \preceq M'$) whenever for every state $s \in I$, there exists a state $s' \in I'$ such that $M, s \preceq M', s'$.

Note that in this definition, M' may have a smaller set of visible state components than M. In this case, we view A' as being the externally visible state components, and $A - A'$ as internal components.

Example 3.7 Consider the countdown timer example (example 3.4). The structure corresponding to the Moore machine of figure 3.4 is given in figure 3.9. We will denote this structure by M. The structure M'

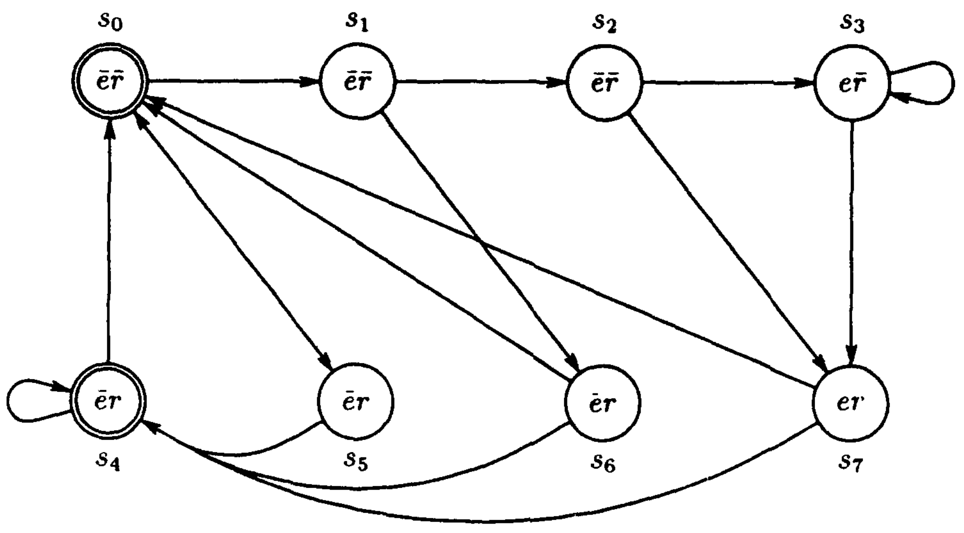
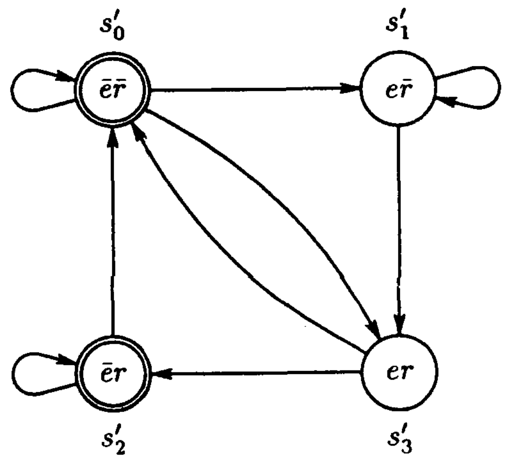


Figure 3.9: Structure for a countdown timer

for the abstract model of a countdown timer is shown in figure 3.10. We have that $M \preceq M'$. To see this, define the relation $\sqsubseteq$ by $s \sqsubseteq s'$ iff $L(s) = L'(s')$. That is, s'_0 is related to s_0, s_1, and s_2; s'_1 is related to s_3; s'_2 is related to s_4, s_5, and s_6; and s'_3 is related to s_7. This obviously satisfies the first condition for a simulation relation: related states have compatible labelings. All paths in M have corresponding paths in M': for example, the path $s_0 s_1 s_2 s_3 s_7 s_0 s_1 s_2 s_3 s_7 \ldots$ corresponds to $s'_0 s'_0 s'_0 s'_1 s'_3 s'_0 s'_0 s'_0 s'_1 s'_3 \ldots$. Thus, $\sqsubseteq$ is indeed a simulation relation. Notice that because of the acceptance conditions, the sequence $s'_0 s'_0 s'_0 \ldots$ is not a path in M'. This is as expected: there is no path in M where r remains 0 indefinitely and e never becomes 1. Finally, for every initial state of M, there is an initial state in M' that is related under $\sqsubseteq$. □

Example 3.8 Let A be a set of visible state components, and define $\top(A)$ to be the following structure M:

1. S is the set of labeling functions over A.



$\mathbf{GF}(r = 0 \rightarrow e = 1)$

Figure 3.10: Structure for the abstract model of a countdown timer

2. $I = S$.

3. $R = S \times S$.

4. $L(f, a) = f(a)$ for all $f \in S$.

5. $F = \emptyset$.

This structure has one state for every possible valuation of the visible state components. Every state is initial, and there is a transition between any pair of states. Further, the acceptance condition is empty, so all sequences of states are legal paths. The structure $\top(A)$ can simulate any other structure whose visible state components include A.

Define $\bot(A)$ to be the structure M with $S = I = R = F = \emptyset$. Every structure whose state components are contained in A can simulate $\bot(A)$. □

Example 3.9 Let M be a structure, and suppose that we add initial states and transitions to M to obtain M'. (That is, $S = S'$, $I \subseteq I'$ and $R \subseteq R'$.) Then $\{ (s, s) \mid s \in S \}$ is a simulation relation, and $M \preceq M'$. Also, if M has (P, Q) as part of its acceptance condition, then we can drop the entire pair from F', or we can enlarge P and Q (with respect to set inclusion) and still maintain the relationship $M \preceq M'$. □

Example 3.10 Here, we consider a way of hiding internal information in a structure M. Let *collapse*$(s) = L(s)$ for $s \in S$. In other words, *collapse* maps a state to the labeling function for that state. Thus, the only information we have about a state after collapsing is what we can observe directly. We extend *collapse* to sets of states and relations between states in the natural way. Then we take *collapse*(M) to be the following structure M':

1. $S' = \mathit{collapse}(S)$.

2. $I' = \mathit{collapse}(I)$.

3. $R' = \mathit{collapse}(R)$.

4. $A' = A$.

5. $L'(L(s), a) = (L(s))(a)$. (The labeling of a labeling function is given by the labeling function itself.)

6. $F' = \{ (\mathit{collapse}(P), \mathit{collapse}(Q)) \mid (P, Q) \in F \}$.

Now $\{ (s, L(s)) \mid s \in S \}$ is a simulation relation between states of M and states of M', and $M \preceq M'$.

As an example of this type of collapsing, let M be the countdown timer of figure 3.9. When we collapse M, we obtain the structure shown in figure 3.11. This is almost the same as the abstract countdown timer model (figure 3.10); the only difference is that collapsing leaves us with an empty acceptance condition. □

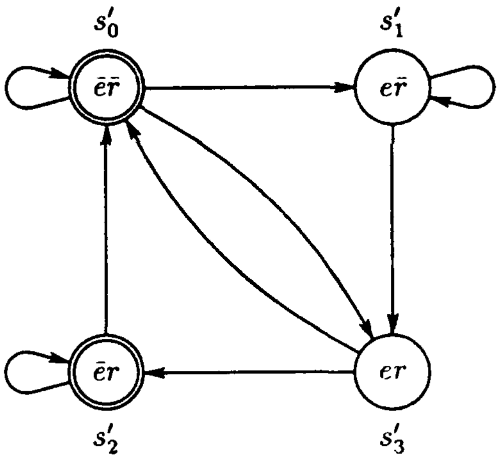


Figure 3.11: Collapsing of the structure for a countdown timer

We now examine some of the properties of the relation $\preceq$. Most of these were mentioned earlier in section 3.2. The proofs of these properties will be deferred; here, we will just try to give the intuition behind each proof.

The first property tells us that the relation $\preceq$ between states is itself a simulation relation, and is in fact the largest simulation relation (under the set inclusion ordering).

Theorem 3.1 Let M and M' be two structures with $A \supseteq A'$. The relation $\preceq$ (between states) is the largest simulation relation between M and M' (under the set inclusion ordering).

To see this, imagine that we have two states s and s' that are related by $\preceq$. By definition, we must have some simulation relation $\sqsubseteq$ that relates the states. This implies that the states have compatible labelings. Further, if we look at a path π from s, there must be some path from s' that corresponds to π via $\sqsubseteq$. But since corresponding states on the two paths are related by $\sqsubseteq$, they must also be related by $\preceq$. Hence $\preceq$ satisfies the conditions for a simulation relation.

The next property forms the basis for doing hierarchical verification. The important part is that $\preceq$ is a transitive relation.

Theorem 3.2 The relation $\preceq$ is a preorder.

Reflexivity is obvious. For transitivity, suppose we know that $M \preceq M'$ and $M' \preceq M''$. Intuitively, if we take a state s in M, then we should be able to find a state s' in M' that simulates it. Then this state can be simulated by some state s'' of M''. Now the labelings of s and s'' must clearly be compatible. Further, given a path from s, we can find a corresponding path from s', and this latter path has a corresponding path from s''. This gives us a correspondence between paths from s and paths from s''. Formally, we would show that $\sqsubseteq$ defined by:

$$\sqsubseteq \; = \{\, (s, s'') \mid \exists s' [s \preceq s' \wedge s' \preceq s''] \,\}$$

is a simulation relation.

Next, we prove that composition respects the preorder. This is used to substitute specifications for implementations in compositions.

Theorem 3.3 For all structures M and M', if $M \preceq M'$, then $M \| M'' \preceq M' \| M''$.

To see why this is true, consider a state (s, s'') of $M \| M''$. Since M' can simulate M, there should be some state s' that can simulate s. Now, since the labelings of s and s' must be compatible, (s', s'') must be a state of the composition $M' \| M''$. Given a path in $M \| M''$ from (s, s''), we can project this down into paths π and π'' in M and M'', respectively. Now π can be simulated by a path π' from s', and π' and π'' can be combined into a path in $M' \,\|\, M''$ from (s', s''). Formally, we prove that

$$\sqsubseteq = \{ ((s, s''), (s', s'')) \mid s \preceq s' \wedge s'' \in S'' \}$$

is a simulation relation.

The final property that we will use is slightly less intuitive than the others. It essentially states that composition with a structure M is idempotent, i.e., doing it more than once has no effect. Perhaps the best way to think of this is as follows: view a structure M as specifying in some way a set of allowed behaviors. Composition with M is essentially intersecting with this set. Once we have done this, intersecting again will obviously result in no change.

Theorem 3.4 For every structure M, $M \preceq M \,\|\, M$.

The proof of this one is simple: we just note that $\{ (s, (s, s)) \mid s \in S \}$ is a simulation relation.

Now that we have finished defining our preorder and notion of composition, we can be more precise about how to do compositional and assume-guarantee style reasoning in our framework. Recall that in section 2.5, we defined the notion of the maximal closing environment for a Moore machine, and also argued that $M \,\|\, E(M)$ was isomorphic to *struct*(M). We have an analogous result when dealing with structures alone. Note that while there is no notion of input and output, and hence no real notion of a closed system, there is a natural maximal environment for a structure M. Also recall the structure $\top(A)$ defined in example 3.8; $\top(A)$ is able to simulate any structure whose visible state components include those in A. Suppose that M is viewed as a component, and say that the environment M' will interact with it via some state components $B \subseteq A$. Then, since A' includes B, we know that $\top(B)$ can simulate M'. Now applying theorem 3.3, we find that $M \,\|\, M' \preceq M \,\|\, \top(B)$. Hence, if we want to check that a specification M'' is true for any environment M', we can just verify $M \,\|\, \top(B) \preceq M''$, since transitivity would then give $M \,\|\, M' \preceq M''$. (Also, $\top(B)$ is a potential environment.) However, we also have the following result:

Theorem 3.5 Let M be a structure and $B \subseteq A$; then M is isomorphic to $M \parallel \top(B)$.

The proof is deferred, but basically consists of the observation that each state of M is paired with a unique state of $\top(B)$ in the composition, and that the transition relation of $\top(B)$ is the universal relation. This result means that when we check $M \preceq M''$, where M is viewed as a component, we are really checking whether every system containing M satisfies the specification M''. On the other hand, if M is viewed as a complete system, then we would just be checking that M has the specified property. In essence, doing a compositional verification, where we are working with individual components, will involve the same underlying check as doing a global verification. Because of this, we will generally omit any mention of maximal environments in what follows. They may be inserted where appropriate, depending on whether the structures we are working with are viewed as complete systems or not.

Recall that in an assume-guarantee style proof, we work with triples of the form $\langle assumptions \rangle M \langle guarantees \rangle$. We will allow assumptions and guarantees to be given either via temporal formulas or via structures. ***To check a triple, we compose the assumptions with M, and then check that the result can be simulated by the guarantees.*** Consider a simple assume-guarantee style argument such as the following:

$$\frac{\begin{array}{c}\langle\rangle M \langle M_\varphi \rangle \\ \langle M_\varphi \rangle M' \langle M_\psi \rangle\end{array}}{\langle\rangle M \parallel M' \langle M_\psi \rangle.}$$

Reexpressing this in terms of compositions and simulation checks gives:

$$\frac{\begin{array}{c}M \preceq M_\varphi \\ M_\varphi \parallel M' \preceq M_\psi\end{array}}{M \parallel M' \preceq M_\psi.}$$

We can justify the soundness of the argument using the properties of $\preceq$ and $\parallel$. Since $M \preceq M_\varphi$, we can compose both sides with M' to obtain $M \parallel M' \preceq M_\varphi \parallel M'$. We are given that $M_\varphi \parallel M' \preceq M_\psi$, so by transitivity we have the desired conclusion.

Let us also consider a more complex argument that requires the use of theorem 3.4:

$$\frac{\begin{array}{c}\langle\rangle M \langle M_\varphi\rangle \\ \langle M_\varphi\rangle M' \langle M_\psi\rangle \\ \langle M_\psi\rangle M \langle M_\chi\rangle\end{array}}{\langle\rangle M \parallel M' \langle M_\chi\rangle.}$$

Translating gives:

$$\frac{\begin{array}{c}M \preceq M_\varphi \\ M_\varphi \parallel M' \preceq M_\psi \\ M_\psi \parallel M \preceq M_\chi\end{array}}{M \parallel M' \preceq M_\chi.}$$

As above, $M \preceq M_\varphi$ and $M_\varphi \parallel M' \preceq M_\psi$ implies that $M \parallel M' \preceq M_\psi$. Composing both sides with M leads to $M \parallel M' \parallel M \preceq M_\psi \parallel M$. Now $M_\psi \parallel M \preceq M_\chi$, so by transitivity we obtain $M \parallel M' \parallel M \preceq M_\chi$. Since composition is commutative and associative, we get $M \parallel M \parallel M' \preceq M_\chi$. Now we are almost at the desired conclusion, but we have an extra M. Theorem 3.4 tells us that $M \preceq M \parallel M$. Composing both sides of this with M': $M \parallel M' \preceq M \parallel M \parallel M'$. Finally, applying transitivity gives the desired result, $M \parallel M' \preceq M_\chi$.

3.5 The Tableau Construction

So far, we have defined notions of composition of structures and simulation that allow us to do hierarchical and assume-guarantee style reasoning where the specifications are given as structures. We have also hinted that there is a correspondence between simulation and satisfaction of a formula by a structure; in this section, we make that correspondence precise.

Our tableau construction will have the same flavor as many others: states of the tableau will consist of information about the labeling for the visible state components, plus information about what things should hold in successor states [5, 27, 64, 78, 92]. This latter information is used to constrain the transition relation.

Example 3.11 Consider the formula $\mathbf{AF}\, a = 1$ that we used earlier (example 3.2). A state of this tableau will be viewed as consisting of:

1. information about whether a is 0 or 1; and

2. information about whether the eventuality has been fulfilled yet or not. This information is based on the fixed point equation for **AF**: those states where $\mathbf{AF}\, a = 1$ are true are those for which $a = 1$ or for which $\mathbf{AX}\,\mathbf{AF}\, a = 1$ holds. We can tell whether $a = 1$ based on the visible state component information. However, in order to tell whether $\mathbf{AX}\,\mathbf{AF}\, a = 1$ holds, we add a bit to the state that will be 1 for those states satisfying $\mathbf{AX}\,\mathbf{AF}\, a = 1$.

Now a state that has the bit for $\mathbf{AX}\,\mathbf{AF}\, a = 1$ set will be constrained to have successors that either have $a = 1$ or have the bit for $\mathbf{AX}\,\mathbf{AF}\, a = 1$ set. □

The information about what has to hold in the next state is captured using the notion of an *elementary formula*. Each elementary formula will have the form $\mathbf{AX}\,\psi$ and will be associated with a bit of information in the states of our tableau. When the bit associated with $\mathbf{AX}\,\psi$ in a state is 1, it will mean that the successors of that state must be constrained to be those states where ψ holds. The elementary formulas of a formula φ will be obtained by looking at those subformulas of φ involving a temporal operator. Each subformula $\mathbf{AX}\,\psi$ will itself be an elementary formula. For subformulas such as $\mathbf{A}(\psi\ \mathbf{U}\ \chi)$, we will use $\mathbf{AX}\,\mathbf{A}(\psi\ \mathbf{U}\ \chi)$ as an elementary formula. In the example above, $\mathbf{AX}\,\mathbf{AF}\, a = 1$ would be the only elementary formula of $\mathbf{AF}\, a = 1$.

Definition 3.5 The *set of elementary formulas of the formula* φ, denoted by $el(\varphi)$, is defined as follows:

1. $el(\mathit{true}) = \emptyset$.

2. $el(a = d) = \emptyset$.

3. $el(\neg\varphi) = el(\varphi)$.

4. $el(\varphi \wedge \psi) = el(\varphi \vee \psi) = el(\varphi) \cup el(\psi)$.

5. $el(\mathbf{AX}\,\varphi) = \{\mathbf{AX}\,\varphi\} \cup el(\varphi)$.
   $el(\mathbf{A}(\varphi\ \mathbf{U}\ \psi)) = \{\mathbf{AX}\,\mathbf{A}(\varphi\ \mathbf{U}\ \psi)\} \cup el(\varphi) \cup el(\psi)$.
   $el(\mathbf{A}(\varphi\ \mathbf{V}\ \psi)) = \{\mathbf{AX}\,\mathbf{A}(\varphi\ \mathbf{V}\ \psi)\} \cup el(\varphi) \cup el(\psi)$.

The states of the tableau for φ are going to have the form (f, E), where $f \in \mathit{labelings}(\mathit{comp}(\varphi))$ and $E \subseteq \mathit{el}(\varphi)$. That is, a state will be a labeling plus a set of elementary formulas (the ones that are supposed to be true at the state). Now suppose that $\mathbf{AX}\,\psi$ is an elementary formula that is supposed to hold at some state. We want to constrain the successors to be those states where ψ is true. But since we have not yet constructed the transition relation, how do we know which states are supposed to satisfy ψ? At first, it seems that we are caught in a kind of circularity. We will avoid the problem by using a mapping Φ that tells whether a formula *should* be true of a state based only on the state and not on its successors. Consider, for example, determining whether a state satisfies $\mathbf{AF}\,a = 1$. If the labeling of a in the state is 1, we know it satisfies $\mathbf{AF}\,a = 1$. If the labeling of a is 0, then the only way that $\mathbf{AF}\,a = 1$ can be true is for all of the state's successors to satisfy $\mathbf{AF}\,a = 1$. In other words, the state should satisfy $\mathbf{AX}\,\mathbf{AF}\,a = 1$. This, however, is an elementary formula, and we can tell whether it should be true by looking only at the state. Overall, a state should satisfy $\mathbf{AF}\,a = 1$ when it is labeled with $a = 1$ or $\mathbf{AX}\,\mathbf{AF}\,a = 1$. Given a subformula ψ, $\Phi(\psi)$ will give the set of states in the tableau that should satisfy ψ. Then, if a state is marked with the elementary formula $\mathbf{AX}\,\psi$, we simply ensure that all of its successors are within the set $\Phi(\psi)$.

The only part of the construction that we have not yet explained is the method by which we ensure that eventualities are fulfilled. This will be done using the acceptance conditions. Consider the formula $\mathbf{AF}\,a = 1$ again. A state that is supposed to satisfy $\mathbf{AX}\,\mathbf{AF}\,a = 1$ has as its successors those states where $a = 1$ or where $\mathbf{AX}\,\mathbf{AF}\,a = 1$ should hold. The danger is that we may pass continually through states where $a = 0$ but which should satisfy $\mathbf{AX}\,\mathbf{AF}\,a = 1$. We can eliminate this possibility by requiring that infinitely often, we visit a state where $a = 1$ or where $\mathbf{AX}\,\mathbf{AF}\,a = 1$ is not supposed to be satisfied. We now give the construction. (Note: the definition below does not handle certain degenerate cases. Since the changes needed to handle these cases are somewhat nonintuitive, we defer them until section 3.8.)

Definition 3.6 The *tableau of* φ (over a set of state components $A \supseteq \mathit{comp}(\varphi)$) is denoted $\mathcal{T}(\varphi)$ and is the structure M given by:

1. $S = \mathit{labelings}(A) \times 2^{\mathit{el}(\varphi)}$.

2. $I = \Phi(\varphi)$, where Φ is the map from subformulas and elementary formulas of φ to S defined as follows:

   (a) $\Phi(\mathit{true}) = S$.

   (b) $\Phi(a = d) = \{ (f, E) \in S \mid f(a) = d \}$.

   (c) $\Phi(\neg\psi) = S - \Phi(\psi)$.

   (d) $\Phi(\psi \wedge \chi) = \Phi(\psi) \cap \Phi(\chi)$.
   $\Phi(\psi \vee \chi) = \Phi(\psi) \cup \Phi(\chi)$.

   (e) If $\mathbf{AX}\,\psi$ is an elementary formula of φ, then

   $$\Phi(\mathbf{AX}\,\psi) = \{ (f, E) \in S \mid \mathbf{AX}\,\psi \in E \}.$$

   (f) $\Phi(\mathbf{A}(\psi\,\mathbf{U}\,\chi)) = \Phi(\chi) \cup (\Phi(\psi) \cap \Phi(\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi)))$.
   $\Phi(\mathbf{A}(\psi\,\mathbf{V}\,\chi)) = \Phi(\chi) \cap (\Phi(\psi) \cup \Phi(\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{V}\,\chi)))$.

3. $R((f_0, E_0), (f_1, E_1))$ iff for all $\mathbf{AX}\,\psi \in \mathit{el}(\varphi)$, $\mathbf{AX}\,\psi \in E_0$ implies $(f_1, E_1) \in \Phi(\psi)$.

4. $L((f, E), a) = f(a)$.

5. The acceptance condition specifies that we cannot have an eventuality $\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi)$ where χ is never fulfilled.

   $$F = \{ (\emptyset, (S - \Phi(\mathbf{AX}\,\mathbf{A}(\psi\mathbf{U}\chi))) \cup \Phi(\chi)) \mid \mathbf{AX}\,\mathbf{A}(\psi\mathbf{U}\chi) \in \mathit{el}(\varphi) \}.$$

Example 3.12 Back in example 3.1, we argued that assume-guarantee style reasoning could be used to verify that the composition of the circuits given in figures 2.2 and 2.4 satisfied the specification $\mathbf{AG}(p = 0 \vee q = 0)$. Let the structures for these two circuits (shown in figures 3.6 and 3.7, respectively) be denoted by M and M'. In our assume-guarantee proof, we were going to use $\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\,q = 0)$ as an assumption about M', and then prove the desired property by combining this assumption with M:

$$\frac{\begin{array}{c}\langle\rangle M' \langle \mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\,q = 0)\rangle \\ \langle \mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\,q = 0)\rangle M \langle \mathbf{AG}(p = 0 \vee q = 0)\rangle\end{array}}{\langle\rangle M \parallel M' \langle \mathbf{AG}(p = 0 \vee q = 0)\rangle}.$$

Checking $\langle\rangle M'\langle \mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)\rangle$ will be done with our standard model checking techniques. However, in order to check

$$\langle \mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)\rangle M \langle \mathbf{AG}(p = 0 \vee q = 0)\rangle,$$

we need to construct the tableau for $\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)$ and compose it with M. The states of the tableau will have valuations for r and q, plus information about the elementary formulas. In this case, there are two elementary subformulas: $\mathbf{AX}\,\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)$ and $\mathbf{AX}\, q = 0$. Let ψ be the first of these, and let χ be the second. The (reachable states of the) tableau are shown in figure 3.12. In the figure, the states are labeled with ψ and χ to indicate where these elementary subformulas are true, even though ψ and χ are not actually visible state components. Also, most of the transitions between states are present, so for clarity, the figure uses dashed lines to show which transitions are *missing*. The initial states are those in $\Phi(\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0))$. This is equal to

$$\Phi(r = 0 \rightarrow \mathbf{AX}\, q = 0) \cap (\Phi(\mathit{false}) \cup \Phi(\mathbf{AX}\,\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0))).$$

(The $\Phi(\mathit{false})$ comes from the fact that $\mathbf{AG}\,\psi$ is an abbreviation for $\mathbf{A}(\mathit{false}\ \mathbf{V}\ \psi)$.) Evaluating this expression yields those states (f, E) where:

1. either $f(r) = 1$ or $\mathbf{AX}\, q = 0$ (χ) is in E; and
2. $\mathbf{AX}\,\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0))$ (ψ) is in E.

This is all of the states shown in the figure. Further, since $\psi \in E$ for all of these states, all their successors must be in $\Phi(\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0))$, i.e., we cannot leave the set of states shown. This is how the $\mathbf{AG}$ is continually enforced. Also note that the transitions that are missing are those from states where χ should be true (the lower four states) to those where q is 1 (the leftmost three states). This enforces the constraint that when a state should satisfy $\mathbf{AX}\, q = 0$, it in fact does. After constructing the tableau and doing the model checking, we find that

$$\langle \mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)\rangle M \langle \mathbf{AG}(p = 0 \vee q = 0)\rangle$$

does indeed hold, and so we can in fact conclude

$$\langle\rangle M \parallel M' \langle \mathbf{AG}(p = 0 \vee q = 0)\rangle,$$

which is the desired result. □

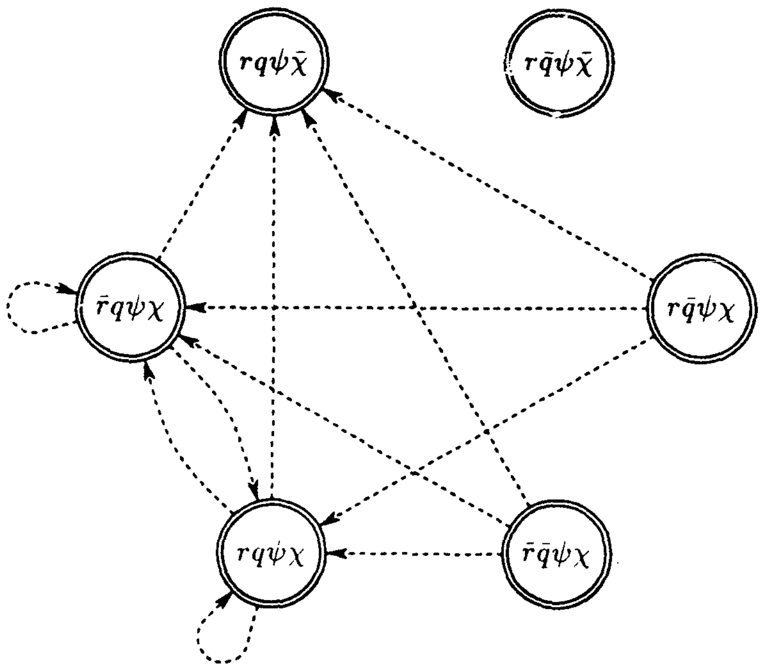
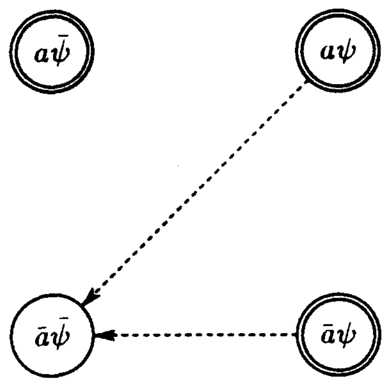


Figure 3.12: Tableau for $\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}\, q = 0)$

Example 3.13 In this example we consider a tableau that has a non-trivial acceptance condition. The actual tableau for $\mathbf{AF}\, a = 1$ is shown in figure 3.13. In the figure, ψ denotes the elementary formula $\mathbf{AX}\,\mathbf{AF}\, a = 1$. As in figure 3.12, only the missing transitions are shown. In this case, we cannot go from a state where $\mathbf{AX}\,\mathbf{AF}\, a = 1$ holds to one where both $a = 0$ and $\neg\, \mathbf{AX}\,\mathbf{AF}\, a = 1$. The acceptance condition requires that if $\mathbf{AX}\,\mathbf{AF}\, a = 1$ becomes true, then eventually we must make a transition to a state where $a = 1$. □

We now state the formal connection between satisfaction and simulation. The proof is deferred.



$\mathbf{GF}(a = 1 \vee \bar{\psi})$

Figure 3.13: Tableau for $\mathbf{AF}\, a = 1$

Theorem 3.6 Let M be a structure and let φ be an ACTL formula such that $A \supseteq comp(\varphi)$. Then $M \models \varphi$ iff $M \preceq \mathcal{T}(\varphi)$, where the tableau is over any subset of A containing $comp(\varphi)$.

We also note that the tableau construction can also be used to do temporal reasoning. In ACTL, $\varphi \rightarrow \psi$ is generally not a legal formula due to the restriction that we only use the **A** path quantifier. Thus, we cannot use the usual trick of checking whether $\varphi \rightarrow \psi$ is a tautology. Instead, we use a semantic notion of entailment.

Definition 3.7 Let φ and ψ be ACTL formulas. We write $\varphi \models \psi$ whenever for every structure M with $A \supseteq comp(\varphi) \cup comp(\psi)$, if $M \models \varphi$, then $M \models \psi$.

The formula φ is a tautology iff $true \models \varphi$. φ is satisfied by some nontrivial structure (one with a non-empty set of initial states and some path starting at one of these states) iff it is not the case that $\varphi \models \mathbf{AX}\, false$. We can check for semantic entailment using the tableau construction in the obvious way.

Proposition 3.2 $\varphi \models \psi$ iff $\mathcal{T}(\varphi) \models \psi$. (The tableau is over $comp(\varphi) \cup comp(\psi)$.)

Since the proof is short, we give it here.

Proof Suppose $\varphi \models \psi$. The tableau for φ satisfies φ, so by definition of semantic implication, it satisfies ψ as well.

Suppose $\mathcal{T}(\varphi) \models \psi$, and let M be a structure with $A \supseteq comp(\varphi) \cup comp(\psi)$. If $M \models \varphi$, then $M \preceq \mathcal{T}(\varphi)$. But $\mathcal{T}(\varphi) \models \psi$, and so $\mathcal{T}(\varphi) \preceq \mathcal{T}(\psi)$. By transitivity, $M \preceq \mathcal{T}(\psi)$, and so $M \models \psi$. □

3.6 Example: A Simple CPU Controller

In this section, we describe a controller for a simple stack-based CPU and give some ACTL specifications describing its correctness. Then we prove these properties using assume-guarantee style reasoning. This CPU controller design is from a paper by Clarke, Long, and McMillan [32].

Figure 3.14 gives a block diagram of the CPU. The controller contains two main modules: an access unit (AU) and an execution unit (EU). The access unit controls the fetching of instructions and the reads and writes to data memory. Instructions are prefetched and stored in an instruction queue (IQ), so that the execution unit will spend less time waiting for instructions to be obtained from memory. The AU also maintains a top-of-stack register (TS) that caches the memory word corresponding to the current stack pointer. Words that are pushed on the stack are stored in this register and flushed to memory when time permits. Similarly, a pop instruction can use the contents of this register without waiting for memory; while this is happening, the TS register is refilled. The execution unit is actually in charge of interpreting the instructions. Our specification will deal mainly with properties of the AU part of the controller, so we will not discuss the EU in detail. We now turn to the signals used by the AU to communicate with its environment. These signals will be used when we give the formal specification later.

The access unit communicates directly with the execution unit via a set of eight lines. Four run from the execution unit to the access unit: *push*, *pop*, *fetch* and *branch*. These signals are used by the EU to express its request to perform the indicated operation. The *push* and

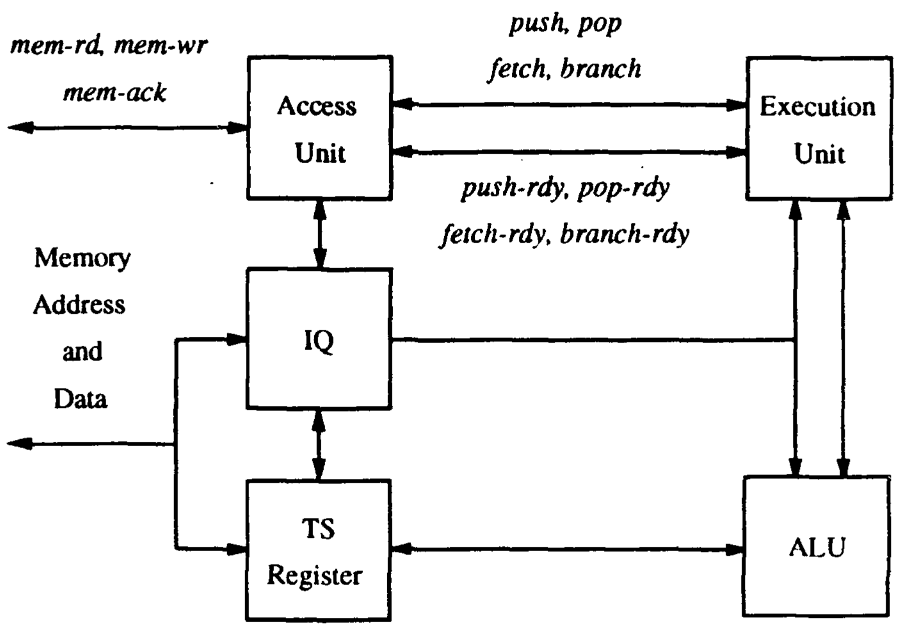


Figure 3.14: CPU block diagram

pop signals are used to manipulate the stack, and the *fetch* signal is u.ed to get the next instruction from the IQ. The EU uses the ***branch*** signal to tell the AU that it wants to execute a (conditional or unconditional) branch and that the instruction queue should be flushed and refilled starting at the new program counter (PC) value. Each of these signals has a corresponding acknowledgment going from the AU to the EU: ***push-rdy***, ***pop-rdy***, ***fetch-rdy*** and ***branch-rdy***. When, e.g., ***push*** and ***push-rdy*** are both high, a word is pushed on the stack. The AU may assert these ready signals before the EU requests the corresponding operation; they are used by the AU to indicate its ability to perform the indicated action immediately.

The access unit also has outputs that control memory reads and writes, and that go to elements of the data path such as the PC and TS registers. The signals ***mem-rd*** and ***mem-wr*** are set high to indicate that a memory read or memory write should be performed. The word to be placed on the memory address lines is signaled by *SP-to-mem-a* and *PC-to-mem-a*; these drive the stack pointer and program counter onto the address lines, respectively. The top of stack register is driven onto the memory data lines using *TS-to-mem-d*. Data coming from memory can be gated into the TS register or into the IQ via *mem-d-to-TS* and *mem-d-to-IQ*. The memory signals completion of a requested operation using the ***mem-ack*** input. To execute a memory cycle, the AU simultaneously asserts ***mem-rd*** or ***mem-wr*** together with one of the signals controlling the memory address bus. When writing, it also asserts *TS-to-mem-d* to drive the data bus. When executing a read, it directs the data into either the TS register or the IQ. It holds these signals until ***mem-ack*** is asserted, then it lowers its control signals and proceeds.

Machine instructions are eight bits long, and two are packed into each sixteen bit machine word. The IQ holds one word which is fetched from an even-aligned address. Hence, when an instruction corresponding to an odd program counter address is used by the EU, the IQ must be refilled. The low bit of the program counter, PC_0, is available to the AU so that it can detect this situation.

The model of the CPU controller is given in the hardware description language CSML (Compositional State Machine Language) [32]. CSML is an extension of the SML language [14] and is designed to support the modular design of finite-state controllers. It provides a module facility to augment SML's procedural description constructs. From the point of view of our verification techniques, the important feature is that its output is a series of Moore machines, one per state machine in the design. We will not go into detail on all the facilities of CSML here. Instead, we will give a simple example, and then proceed to the CSML code for the AU.

Figure 3.15 is a CSML program describing a system composed of a producer module and a consumer module which synchronize using a four-phase handshake. In CSML (as in SML), raising or lowering an externally visible signal takes one time step, i.e., one Moore machine transition occurs. All other computation takes no (external) time. The **`raise`** and **`lower`** statements are used to set and reset signals. The control constructs such as **`while`** and **`loop`** have the obvious meanings. The **`process`** declarations (starting on line 26) actually create the two Moore machines in this example. A **`processtype`** (line 4) is used to give a template for each machine.

We now turn to the CSML description of the AU. The main functions of the AU are managing the TS register and the IQ. The top-of-stack register can conceptually be in one of three states.

1. It may be *invalid*, in which case the EU is allowed to push (store data in TS), but not pop (get data from it).

2. It may be *valid*, meaning that the data in TS matches what is in memory at the address indicated by the SP. In this case, the EU may either push or pop.

3. It may be *modified*, meaning that the EU has placed data in the TS register and that data has not yet been copied out to memory. In this case, then EU is allowed to pop, but it cannot be allowed to execute a push.

The transitions between these states are as shown in the state transition diagram in figure 3.16.

Part of the AU code that is used to control the TS register state is shown in figure 3.17. This code tells how the state changes when the EU executes a push or pop. The **`compress`** statement (line 2) is

```
program prodcom
  output produce,consume;
  internal req,ack;

  processtype producer(request, acknowledge,  produce)
    input request;
    output acknowledge=false, produce=false;
    loop
      while (!request) do loop skip endloop;
      raise(produce); lower(produce);
      raise(acknowledge);
      while (request) do loop skip endloop;
      lower(acknowledge)
    endloop
  endtype

  processtype consumer(acknowledge, request, consume)
    input acknowledge;
    output request=false, consume=false;
    loop
      raise(request);
      while (!acknowledge) do loop skip endloop;
      raise(consume); lower(consume);
      lower (request);
      while(acknowledge) do loop skip endloop
    endloop
  endtype

  process producer1: producer(req, ack, produce);
  process consumer1: consumer(ack, req, consume)
endprog
```

Figure 3.15: Producer-consumer program in CSML

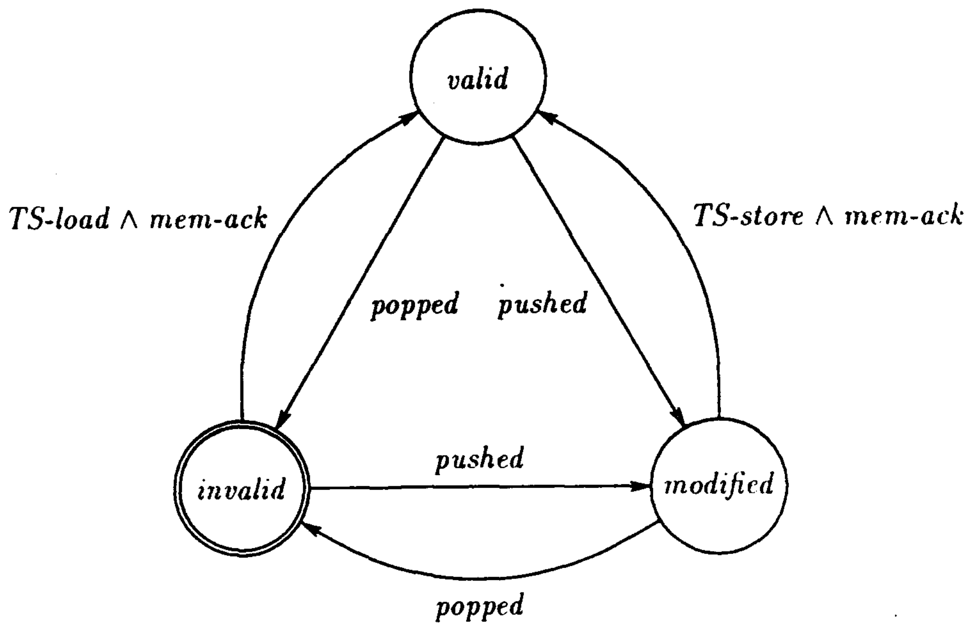


Figure 3.16: TS state transition diagram

used to cause all the state changes to happen in one external time step (one Moore machine transition). The **ts_st** variable is used to hold the current state of the TS register.

```
loop
  compress
    switch
      case ((ts_st == valid) | (ts_st == invalid))
          & push & push_rdy:
        lower(push_rdy);
        raise(pop_rdy);
        ts_st := modified;
        break;
      case ((ts_st == valid) | (ts_st == modified))
          & pop & pop_rdy:
        lower(pop_rdy);
        raise(push_rdy);
        ts_st := invalid;
        break;
      default: skip;
    endswitch
  endcompress
endloop
```

Figure 3.17: CSML code implementing TS control

The other piece of code responsible for setting the TS state is the section in charge of memory accesses. This section is shown in figure 3.18. The second and third elements of the case statement examine the state of the TS register. If it is *invalid* (and the EU is not trying to execute a push), then the AU may load the register from memory (line 12). If the state is *modified* (and the EU does not want to pop), then the TS contents are copied to memory (line 21). This part of the code is also responsible for prefetching instructions (line 3).

The access unit also manages the IQ. This is done in a similar manner to the TS register control (figure 3.17); for brevity, we omit the

```
loop
  switch
    case iq_st == invalid:
      compress
        lower(branch_rdy);
        read(pc_to_mem_a, mem_d_to_iq);
        iq_st := valid;
        raise(fetch_rdy);
        raise(branch_rdy);
      endcompress;
      break;
    case ts_st == invalid & !push:
      compress
        lower(push_rdy);
        read(sp_to_mem_a, mem_d_to_ts);
        ts_st := valid;
        raise(push_rdy);
        raise(pop_rdy)
      endcompress;
      break;
    case ts_st == modified & !pop:
      compress
        lower(pop_rdy);
        write(sp_to_mem_a, ts_to_mem_d);
        ts_st := valid;
        raise(push_rdy);
        raise(pop_rdy)
      endcompress;
      break;
    default: skip;
  endswitch
endloop
```

Figure 3.18: CSML code for controlling memory accesses

actual code. Altogether, the AU is composed of these three threads of control (TS and IQ managers, and memory access manager) running in parallel. When processed by the CSML compiler, the result is a Moore machine with thirteen states.

The execution unit is more complex; it essentially consists of a large case statement, with one case per instruction. We will not give the code here, but it compiles into a Moore machine with 98 states. The combined CPU controller, plus a two state memory model, is a Moore machine with 1077 states.

We now give a formal specification of the AU in ACTL. A formal specification of the EU will not be given; it would consist of a large number of cases (one per instruction). To begin, we will define a few abbreviations that will be used throughout the formulas here. The first ones are used to say that a push, pop, fetch or branch has occurred. Each of them is a conjunction of an EU request signal and an AU acknowledge signal.

$$\begin{aligned}
\mathit{pushed} &= \mathit{push} \wedge \mathit{push\text{-}rdy} \\
\mathit{popped} &= \mathit{pop} \wedge \mathit{pop\text{-}rdy} \\
\mathit{fetched} &= \mathit{fetch} \wedge \mathit{fetch\text{-}rdy} \\
\mathit{branched} &= \mathit{branch} \wedge \mathit{branch\text{-}rdy}
\end{aligned}$$

The next three tell when the IQ is being loaded and when the TS register is being loaded or stored into memory. For example, the IQ is being loaded when the PC is being driven onto the memory data bus, the AU is reading from memory, and the data from memory is being gated into the IQ.

$$\begin{aligned}
\mathit{TS\text{-}load} &= \mathit{mem\text{-}rd} \wedge \mathit{SP\text{-}to\text{-}mem\text{-}a} \wedge \mathit{mem\text{-}d\text{-}to\text{-}TS} \\
\mathit{TS\text{-}store} &= \mathit{mem\text{-}wr} \wedge \mathit{SP\text{-}to\text{-}mem\text{-}a} \wedge \mathit{TS\text{-}to\text{-}mem\text{-}d} \\
\mathit{IQ\text{-}load} &= \mathit{mem\text{-}rd} \wedge \mathit{PC\text{-}to\text{-}mem\text{-}a} \wedge \mathit{mem\text{-}d\text{-}to\text{-}IQ}
\end{aligned}$$

The last one states that the final instruction in the IQ has just been fetched.

$$\mathit{IQ\text{-}emptied} = \mathit{fetched} \wedge \mathit{PC}_0$$

The first class of formulas are some basic safety properties of the access unit. They require that the AU not issue spurious reads and writes, and that each memory access be an IQ or TS load, or a TS store.

$$\mathbf{AG}(\textit{SP-to-mem-a} \rightarrow \textit{TS-load} \vee \textit{TS-store}) \tag{3.1}$$
$$\mathbf{AG}(\textit{mem-d-to-TS} \rightarrow \textit{TS-load}) \tag{3.2}$$
$$\mathbf{AG}(\textit{TS-to-mem-d} \rightarrow \textit{TS-store}) \tag{3.3}$$
$$\mathbf{AG}(\textit{PC-to-mem-a} \rightarrow \textit{IQ-load}) \tag{3.4}$$
$$\mathbf{AG}(\textit{mem-d-to-IQ} \rightarrow \textit{IQ-load}) \tag{3.5}$$
$$\mathbf{AG}(\textit{mem-rd} \rightarrow \textit{TS-load} \vee \textit{IQ-load}) \tag{3.6}$$
$$\mathbf{AG}(\textit{mem-wr} \rightarrow \textit{TS-store}) \tag{3.7}$$

We also cannot allow multiple memory accesses to be attempted at the same time. The AU should not, e.g., drive both *mem-rd* and *mem-wr* high at the same time.

$$\mathbf{AG}(\neg \textit{TS-load} \vee \neg \textit{TS-store}) \tag{3.8}$$
$$\mathbf{AG}(\neg \textit{TS-load} \vee \neg \textit{IQ-load}) \tag{3.9}$$
$$\mathbf{AG}(\neg \textit{TS-store} \vee \neg \textit{IQ-load}) \tag{3.10}$$

Next, we require that if the AU requests a memory operation, then it must continue to request that operation until it receives an acknowledgment. That is, memory requests cannot be aborted in mid-cycle. We can express this using the **V** operator: *mem-ack* will release the requirement that the load or store signals remain stable.

$$\mathbf{AG}(\textit{TS-load} \rightarrow \mathbf{A}(\textit{mem-ack}\ \mathbf{V}\ \textit{TS-load})) \tag{3.11}$$
$$\mathbf{AG}(\textit{TS-store} \rightarrow \mathbf{A}(\textit{mem-ack}\ \mathbf{V}\ \textit{TS-store})) \tag{3.12}$$
$$\mathbf{AG}(\textit{IQ-load} \rightarrow \mathbf{A}(\textit{mem-ack}\ \mathbf{V}\ \textit{IQ-load})) \tag{3.13}$$

Also, the access unit should not offer the EU the chance to push, pop fetch or branch while a memory cycle that might interact with the operation is going on. (These requirements ensure, e.g., that the address being driven onto the memory address bus does not change.)

$$\mathbf{AG}(\textit{TS-load} \vee \textit{TS-store} \rightarrow \neg \textit{push-rdy} \wedge \neg \textit{pop-rdy}) \tag{3.14}$$
$$\mathbf{AG}(\textit{IQ-load} \rightarrow \neg \textit{fetch-rdy} \wedge \neg \textit{branch-rdy}) \tag{3.15}$$

The next set of properties are used to check that the operations allowed by the access unit on the TS register follow the state transition diagram given in figure 3.16. Note that when the state of the TS register is *valid*, then either a push or a pop is legal. Also, while the actual AU does not load or store the top-of-stack in this state, doing so would not cause an error. Hence, we impose no constraints on the actions performed while TS is *valid*. Next, consider the *invalid* state. This state is entered when a pop operation is executed. Starting from this state, we cannot allow another pop, and we cannot store the TS register into memory. The TS register will cease to be *invalid* after the TS register is loaded from memory, or after the EU pushes a word on the stack. Thus, we want to express "after a pop, no pop or TS store can occur until after a TS load or a push." This is done with the following formula:

$$\mathbf{AG}(\mathit{popped} \rightarrow \mathbf{AX}\,\mathbf{A}((\mathit{TS\text{-}load} \wedge \mathit{mem\text{-}ack}) \vee \mathit{pushed} \;\mathbf{V}\; \neg\mathit{pop\text{-}rdy} \wedge \neg\mathit{TS\text{-}store})). \quad (3.16)$$

The TS register should also start out in the *invalid* state, so we obtain the related requirement:

$$\mathbf{A}((\mathit{TS\text{-}load} \wedge \mathit{mem\text{-}ack}) \vee \mathit{pushed} \;\mathbf{V}\; \neg\mathit{pop\text{-}rdy} \wedge \neg\mathit{TS\text{-}load}). \quad (3.17)$$

If the TS register is in the *modified* state (as the result of a push), then both pushes and TS loads are illegal. The TS register state should only change when a pop occurs, or when the TS register contents are stored into memory. We express this requirement with the formula:

$$\mathbf{AG}(\mathit{pushed} \rightarrow \mathbf{AX}\,\mathbf{A}((\mathit{TS\text{-}store} \wedge \mathit{mem\text{-}ack}) \vee \mathit{popped} \;\mathbf{V}\; \neg\mathit{push\text{-}rdy} \wedge \neg\mathit{TS\text{-}load})). \quad (3.18)$$

We now turn to requirements for how the IQ is managed. The IQ can be in one of two states: *valid*, indicating that there is a valid instruction in the queue waiting to be fetched; and *invalid*, indicating that there is no such instruction. Figure 3.19 shows the possible transitions between these states. When the IQ is in the *valid* state, we have no constraints on fetches. The IQ state changes to *invalid* when either a fetch from the last location in the queue or a branch occurs. From

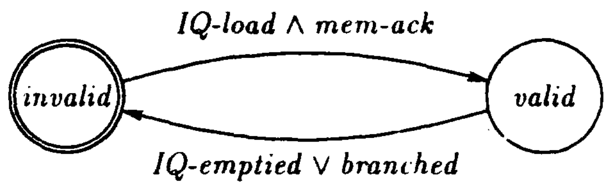


Figure 3.19: IQ state transition diagram

the *invalid* state, no additional fetches can be allowed until the IQ is loaded from memory. Thus, we have the requirement:

$$\mathbf{AG}(\textit{IQ-emptied} \vee \textit{branched} \\ \rightarrow \mathbf{AX}\,\mathbf{A}(\textit{IQ-load} \wedge \textit{mem-ack}\ \mathbf{V}\ \neg\textit{fetch-rdy})). \quad (3.19)$$

The IQ also starts in the *invalid* state, so we also have the related formula:

$$\mathbf{A}(\textit{IQ-load} \wedge \textit{mem-ack}\ \mathbf{V}\ \neg\textit{fetch-rdy}). \quad (3.20)$$

Note that all of the above properties are safety properties; none of them make any guarantees that progress will occur. The following formulas are used to specify that pushes, pops, fetches, and branches always complete.

$$\mathbf{AG}(\textit{push} \rightarrow \mathbf{AF}\,\textit{pushed}) \quad (3.21)$$

$$\mathbf{AG}(\textit{pop} \rightarrow \mathbf{AF}\,\textit{popped}) \quad (3.22)$$

$$\mathbf{AG}(\textit{fetch} \rightarrow \mathbf{AF}\,\textit{fetched}) \quad (3.23)$$

$$\mathbf{AG}(\textit{branch} \rightarrow \mathbf{AF}\,\textit{branched}) \quad (3.24)$$

Finally, we check that the controller continually fetches new instructions:

$$\mathbf{AG}\,\mathbf{AF}\,\textit{fetched} \quad (3.25)$$

We used a BDD-based model checker to verify that the system composed of the AU and EU satisfied the above specification (with some weak assumptions about how the memory behaves). The basic safety properties (formulas 3.1 through 3.15) were checked using the AU alone. As an example, for formula 3.1 we verified:

$$\langle\rangle M_{\mathrm{AU}} \langle \mathbf{AG}(\textit{SP-to-mem-a} \rightarrow \textit{TS-load} \vee \textit{TS-store})\rangle.$$

where M_{AU} is the structure representing the Moore machine for the AU. This implies that every system that can be constructed using the AU satisfies this particular formula.

We also tried using just the AU to verify some of the more complex properties such as formulas 3.16 and 3.18. All of the formulas failed to check, so we examined the error traces produced by the model checker to try to determine the cause of the failure. In all of the traces, the inputs from memory were not behaving as we would have expected in a real system. In particular, the memory acknowledgment signal sometimes went high when there was no pending request. We therefore constructed a model of how the memory was supposed to behave. This model, which we denote by M_{mem}, is shown in figure 3.20. The figure depicts a Moore machine; the actual model used is the corresponding structure.

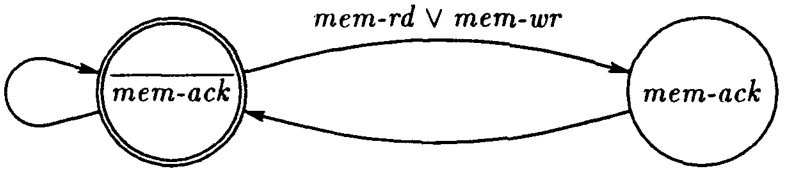

$mem\text{-}rd \lor mem\text{-}wr$
$\overline{mem\text{-}ack}$
$mem\text{-}ack$

Figure 3.20: Memory model

With this model of the memory as an assumption, all of the formulas 3.17 through 3.20 are true. So, for example, we have:

$$\langle M_{\text{mem}}\rangle M_{\text{AU}}\langle \mathbf{A}(\,TS\text{-}load \land mem\text{-}ack \lor pushed\,\mathbf{V}\, \neg pop\text{-}rdy \land \neg\, TS\text{-}load)\rangle.$$

The specification given by 3.16 remains false however. Upon examining the error trace, we find a situation where both the *push-req* and *pop-req* signals become true simultaneously, i.e., the EU attempts both a push and a pop at the same time. This behavior is obviously illegal, so we make another assumption to eliminate it:

$$\mathbf{AG}(\neg push\text{-}req \lor \neg pop\text{-}req). \qquad (3.26)$$

With this assumption plus the assumption M_{mem}, formula 3.16 becomes true.

The only remaining properties to be checked are the progress properties, plus the assumption 3.26. First, we note that in order to be able to ensure progress, the memory must be guaranteed to respond to requests eventually. Consequently, we strengthen our assumption about the memory's behavior by adding acceptance conditions

$$\mathbf{GF}(\mathit{mem\text{-}rd} \wedge \neg\mathit{mem\text{-}wr} \rightarrow \mathit{mem\text{-}ack})$$

and

$$\mathbf{GF}(\mathit{mem\text{-}wr} \wedge \neg\mathit{mem\text{-}rd} \rightarrow \mathit{mem\text{-}ack}).$$

If we now try to check that the AU, plus our assumption about the memory, satisfies the formulas 3.21 through 3.24, we find that it does not. The reason is that the EU may make a request and then immediately remove it without giving the AU time to act. We make additional assumptions about the EU's behavior to eliminate these possibilities.

$$\mathbf{AG}(\mathit{push\text{-}req} \rightarrow \mathbf{A}(\mathit{pushed}\ \mathbf{V}\ \mathit{push\text{-}req})) \tag{3.27}$$

$$\mathbf{AG}(\mathit{pop\text{-}req} \rightarrow \mathbf{A}(\mathit{popped}\ \mathbf{V}\ \mathit{pop\text{-}req})) \tag{3.28}$$

$$\mathbf{AG}(\mathit{fetch\text{-}req} \rightarrow \mathbf{A}(\mathit{fetched}\ \mathbf{V}\ \mathit{fetch\text{-}req})) \tag{3.29}$$

$$\mathbf{AG}(\mathit{branch\text{-}req} \rightarrow \mathbf{A}(\mathit{branched}\ \mathbf{V}\ \mathit{branch\text{-}req})) \tag{3.30}$$

We now attempt to verify property 3.21 for the AU, the memory model, and the assumption 3.27. Again, the formula turns out to be false; in this case, the problem is the EU issuing simultaneous requests. Earlier, we used an assumption that push and pop requests were mutually exclusive (formula 3.26). We strengthen this assumption so that it states that every pair of operations requested by the EU must be mutually exclusive. The weaker assumption can be discharged using the stronger one; we simply check semantic implication between the two formulas. Now using the AU, the memory model, the assumption 3.27, and the mutual exclusion assumption, we are finally able to verify formula 3.21. Similarly, we can verify each of the other liveness properties (through 3.24).

Now we have to check the final liveness property (3.25), plus the assumptions that we made about the behavior of the EU. The assumptions about the behavior of the EU can be checked using just the EU, so we successfully discharge them. As for formula 3.25, there are two approaches that we could use. The first would be to make some additional assumptions about the EU and check the property using the AU and these assumptions. We would need to know that the EU does not fetch an instruction and then execute an infinite sequence of pushes, pops, or branches. To express this, we could build an abstract model of the EU such as the one shown in figure 3.21. This figure shows a Moore machine, but the actual model would be the corresponding structure plus the indicated acceptance condition. In the figure, *push-req* has been abbreviated to *push*, etc., and *idle* indicates that *push-req*, *pop-req*, *fetch-req*, and *branch-req* are all low.

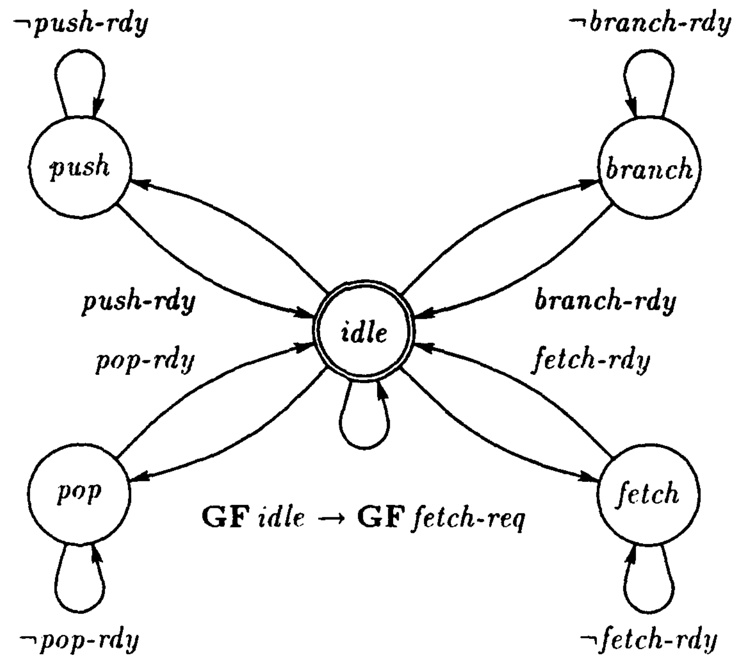


Figure 3.21: Execution unit model

The other possibility would be to try to check the property on the EU. In this case, we would need to know that pushes, pops, etc., eventually complete. However, we have already verified these conditions in properties 3.21 through 3.24. Using these properties as assumptions together with the EU is indeed sufficient to prove **AG AF** *fetched*. In summary then, we have managed to verify all of the properties. The more complex parts of the specification required us to make an assumption about how the memory behaved. Given an actual memory system design, we would need to check that our model of the memory (figure 3.20, plus an acceptance condition) could in fact simulate the design.

3.7 Summary

We have provided a way of doing assume-guarantee style reasoning in the context of ACTL model checking. In order to do this, we first made explicit the important notion behind theorem 2.2: that of simulation. Simulation is a natural relationship between implementation and specification. It leads directly to the ability to do hierarchical verification: specifications at one level become "implementations" at the next. By examining how simulation relates to composition, we were able to give methods for compositional and assume-guarantee style reasoning. However, we already had one notion of satisfaction of a specification, $\models$. Via a tableau construction, we proved that satisfaction of ACTL formulas corresponds directly to simulation. This link gives us great flexibility as to our specification methodology when performing assume-guarantee proofs or doing hierarchical reasoning. We demonstrated these ideas by verifying some properties of the controller for a simple stack-based CPU. Further, the general framework discussed in section 3.2 can be used to construct assume-guarantee style reasoning systems based on other temporal logics.

3.8 Technical Details

In our framework, specifications and assumptions can be given as either formulas or structures. In the latter case, however, we need methods for automatically checking whether one structure simulates another one: that is the subject we now consider. We describe two special case methods that, in practice, cover most of the cases that arise. Further, these special case methods are generally much more efficient than a fully general algorithm.

We have already seen one method; when we are given that the structure M' is the tableau for a formula φ, we can check $M \preceq M'$ by verifying $M \models \varphi$ using the standard model checking algorithm for ACTL. While the model checking algorithm can detect when $M \not\preceq M'$, this fact alone is not very useful; rather, we would like to demonstrate explicitly why this is the case. That is, we want to produce a *counterexample* illustrating why the formula is false. Consider the problem of demonstrating why φ is false at the state s. We break the task into cases based on the top-level operator of φ.

1. If φ is an atomic formula (or the negation of an atomic formula), we can just say why it is inconsistent with the labeling of s.

2. If φ has the form $\psi \wedge \chi$, then at least one of ψ and χ must be false at s. We call the counterexample procedure recursively for the appropriate subformula. Dealing with disjunctions $\psi \vee \chi$ is similar, but we have to demonstrate that both ψ and χ are false at s.

3. If $\varphi = \mathbf{AX}\,\psi$, then we find a successor s_1 of $s = s_0$ that is the start of some path and for which ψ is false at s_1. (The states that are the start of a path can be found using a standard fixed point computation.) We display the "path" (actually the prefix of a path) $s_0 s_1 \ldots$ and then show why s_1 does not satisfy ψ.

4. When φ has the form $\mathbf{A}(\psi \mathbf{V} \chi)$, then there must be a path $s_0 s_1 s_2 \ldots$ starting at $s = s_0$ for which:

   (a) χ is false at some s_i; and

   (b) for all $j < i$, ψ is false at s_j.

   Starting from s_0, we search forward to find such a path. We will compute a series of sets P_i where P_i represents the search frontier after stepping forwards i times. We begin with $P_0 = \{s_0\}$. After computing P_i, we see whether there are any states in P_i that are the start of some path and that do not satisfy χ. If so, then we have found one (or perhaps several) s_i satisfying condition 4a above. We select one such s_i; now we must back up to produce a path to s_0. We see which states in P_{i-1} can reach s_i in one step. Now s_{i-1} is chosen from these states, and we proceed backwards until we eventually reach s_0.

Suppose now that every state in P_i satisfies χ. In this case, we must search forward another step. However, we also need to be sure that we do not pass through a state satisfying ψ. Thus, we let Q_i be the states in P_i that do not satisfy ψ. We then define P_{i+1} to be the states reachable by stepping forwards once from Q_i.

The above procedure gives us a (prefix of a) path $s_0 s_1 \ldots s_i \ldots$ where s_i does not satisfy χ and each s_j for $j < i$ does not satisfy ψ. We display this prefix, then call the counterexample facility recursively to show why ψ is false at the s_j $(j < i)$ and why χ is false at s_i.

5. The most interesting case is for formulas of the form $\mathbf{A}(\psi \mathbf{U} \chi)$. Such a formula may be false for one of two reasons.

   (a) There may be a path $s_0 s_1 s_2 \ldots$ from $s = s_0$ such that ψ is false at some s_i, and for all $j \leq i$, χ is false at s_j. We can determine whether there is such a path (and if so, display it) using the same techniques as above.

   (b) There may be a path $s_0 s_1 s_2 \ldots$ from $s = s_0$ such that χ is false at every state on this path. (That is, the eventuality is never fulfilled.) This is the case we now consider.

Obviously, we cannot construct or display arbitrary infinite paths. Instead, we will find finite sequences of states π_0 and π_1 such that $\pi_0 \pi_1^\omega$ (π_0 followed by infinite repetitions of π_1) is a path from s satisfying these constraints. We can then display π_0 and π_1 and show why χ is false at every state appearing in π_0 or π_1.

The question now is how to find such a pair of sequences. We will do this by trying to find a *fair strongly connected component* (FSCC). A strongly connected component (SCC) is a set of states where each state in the set can reach every other state in the set via the transition relation. For every state, there is some SCC that contains it (a singleton set is an SCC), and there is a unique *maximal SCC* (under the set inclusion ordering) containing the state. An SCC is fair if it contains some path that stays entirely within the SCC. We only want to consider states where there is an infinite path along which χ is false, so we first eliminate all states not satisfying $\mathbf{EG}\,\neg\chi$ from the structure. (Note that s is in the result.) Next, we compute the maximal SCC C containing s. We then check to see whether C is an FSCC (this can be done using a standard fixed point computation). If it is not, then since s is the start of some path along which χ is false, we know that there must be a sequence of transitions from s leading out of C to a state s' that satisfies $\mathbf{EG}\,\neg\chi$. We then find the maximal SCC containing s', test if this SCC is an FSCC, and, if necessary, repeat the process. Eventually, we must find a state reachable from s for which the maximal SCC is an FSCC. The sequence of transitions from s to this state gives us π_0, the prefix of the infinite path that we are constructing.

Now we need to find a loop within the FSCC such that each pair (P, Q) in the acceptance condition is satisfied along this loop. Recall that Q represents the "infinitely often" part of the constraint. Let C denote the FSCC and without loss of generality, assume that C is the SCC for s. Let us first consider the case where for every pair in the acceptance condition, Q intersects C. In this case, we can simply choose a state from each intersection, visit these states in some order, and then return to s. The result is a loop containing s along which some state in each Q is visited. If we let π_1 be the sequence of states encountered in going once around this loop, then clearly π_1^ω is a path. Further, we restricted ourselves earlier to those states satisfying $\mathbf{EG}\,\neg\chi$, so we have found a path along which χ remains false.

Suppose now that for some of the pairs (P, Q) in the acceptance condition, Q does not intersect C. In order to satisfy such a pair, we must have a loop where each state on the loop is in P. Since s may not be in P, we cannot necessarily find a loop containing s. Thus, we may have to extend the prefix π_0. Our goal will be to find an FSCC C' within $C \cap P$, then append a segment to π_0 that takes us from s to C'. Then we will find a loop within C'; note that this entire loop will have to be in P. Thus, we can eliminate (P, Q) from consideration. Eventually, we will either eliminate all pairs (in which case any loop will satisfy all of the "almost always" conditions), or we will be able to satisfy all of the remaining pairs using the "infinitely often" parts. To find C', we let D be the intersection of C with P and then determine which states in D are the start of a path that stays entirely within D. Let D' be these states; we restrict our attention only to D'. We choose one of these states, and then find its SCC (within D'). If this SCC is an FSCC, we have found C'. Otherwise, we simply choose a different state of D'.

The other situation that arises most often in practice is for M' to be *deterministic*. By this, we intuitively mean that there is no state which has transitions to two successors with the same labeling. (However, note that there may be multiple states with the same labeling.)

Definition 3.8 M is *deterministic* if:

1. For all s_0 and s_1 in I (with $s_0 \neq s_1$), $L(s_0) \neq L(s_1)$.
2. For all $s \in S$, if $R(s, s_0)$ and $R(s, s_1)$ (with $s_0 \neq s_1$), then $L(s_0) \neq L(s_1)$.

When M' is deterministic, given states s and s' and a path π from s, there is only one possible path from s' that could correspond to π. Thus, in this case, $\preceq$ essentially corresponds to *ω-language containment.*

Definition 3.9 The *language of M, s_0 over a set of observable state components* $A' \subseteq A$ (denoted by $\mathcal{A}(M, s_0, A')$) is the set of sequences of labelings occurring on paths starting from s_0.

$$\mathcal{A}(M, s_0, A') = \{ f_0 f_1 f_2 \ldots \mid s_0 s_1 s_2 \ldots \text{ is a path}, \forall i\, f_i = L(s_i) \downarrow A' \}$$

(Recall that $L(s_i) \downarrow A'$ denotes $L(s_i)$ with its domain restricted to A'.) We write $\mathcal{A}(s, A')$ when M is understood. The language of M is the union of the languages for all of its initial states.

$$\mathcal{A}(M, A') = \bigcup_{s \in I} \mathcal{A}(M, s, A').$$

Proposition 3.3 We have the following relationship between language containment and simulation:

1. If $M \preceq M'$, then $\mathcal{A}(M, A') \subseteq \mathcal{A}(M', A')$ and for every $s \in I$, there is some $s' \in I'$ such that $L(s) \downarrow A' = L'(s')$.

2. Suppose $\mathcal{A}(M, A') \subseteq \mathcal{A}(M', A')$, M' is deterministic, and for every $s \in I$, there is some $s' \in I'$ such that $L(s) \downarrow A' = L'(s')$; then $M \preceq M'$.

The above relationship means that when M' is deterministic, we can check $\preceq$ by basically checking for language containment. This can be done in polynomial time using standard techniques [26].

Proof Assume $M \preceq M'$. Let $\pi = s_0 s_1 s_2 \ldots$ be a path from $s_0 \in I$ in M. There must exist a path $\pi' = s'_0 s'_1 s'_2 \ldots$ from some $s'_0 \in I'$ in M' for which $s_i \preceq s'_i$ for all i. Since $s_i \preceq s'_i$, we have $L(s_i) \downarrow A' = L'(s'_i)$. Hence the sequences of labelings corresponding to π and π' are the same, and so $\mathcal{A}(M, A') \subseteq \mathcal{A}(M', A')$. Obviously, for every $s \in I$, there is some $s' \in I'$ such that $s \preceq s'$, and hence $L(s) \downarrow A' = L'(s')$.

Suppose $\mathcal{A}(M, A') \subseteq \mathcal{A}(M', A')$ and that M' is deterministic. Let $\sqsubseteq$ be the relation

$$\{ (s, s') \mid L(s) \downarrow A' = L'(s') \wedge \mathcal{A}(s, A') \subseteq \mathcal{A}(s', A') \}.$$

We show that $\sqsubseteq$ is a simulation relation; suppose $s \sqsubseteq s'$. By definition, s and s' agree on the labels of state components in A'. Let $\pi = s_0 s_1 s_2 \ldots$ be a path from $s = s_0$ in M. Since $\mathcal{A}(s, A') \subseteq \mathcal{A}(s', A')$, there is a path $\pi' = s'_0 s'_1 s'_2 \ldots$ from $s' = s'_0$ for which the labelings on the two paths (with respect to A') agree. Since $\mathcal{A}(s_0, A') \subseteq \mathcal{A}(s'_0, A')$ and M' is deterministic, $\mathcal{A}(s_1, A') \subseteq \mathcal{A}(s'_1, A')$. Also, $L(s_1) \downarrow A' = L'(s'_1)$; hence $s_1 \sqsubseteq s'_1$. Applying the above argument inductively, we find $s_i \sqsubseteq s'_i$ for all i.

Suppose now $s \in I$. By hypothesis and the fact that M' is deterministic, there is a unique $s' \in I'$ such that $L(s) \downarrow A' = L'(s')$. If there is no path starting at s, then clearly $s \preceq s'$. If there are paths from s, then $\mathcal{A}(M, A') \subseteq \mathcal{A}(M', A')$ implies that $\mathcal{A}(s, A') \subseteq \mathcal{A}(s', A')$. Then $s \sqsubseteq s'$, and so $s \preceq s'$. Thus in all cases, $s \preceq s'$, and so $M \preceq M'$. □

We now turn to some of the proofs that were previously deferred. First, we note that the composition operation for structures is commutative and associative.

Theorem 3.7 Let M, M' and M'' be structures. Then $M \parallel M'$ is isomorphic to $M' \| M$. Also $M \| (M' \| M'')$ is isomorphic to $(M \| M') \| M''$.

Proof For commutativity, it is easy to see that the map taking the state (s, s') of the former to (s', s) of the latter preserves initial states, transitions, labelings, and acceptance conditions, and hence is an isomorphism.

For associativity, let $M_0 = (M \parallel M') \parallel M''$ and $M_1 = M \parallel (M' \parallel M'')$. Let ϕ be the map taking $((s,s'),s'')$ to $(s,(s',s''))$. In order to show that this is a bijection, we need to prove that $((s,s'),s'')$ is a valid state of M_0 iff $(s,(s',s''))$ is a valid state of M_1. Assume $((s,s'),s'') \in S_0$. To show $(s,(s',s'')) \in S_1$, we must first prove that (s',s'') is a state of $M' \parallel M''$. If (s',s'') is not a state of $M' \parallel M''$, then there must be some state component a' in $A' \cap A''$ such that $L'(s',a') \neq L''(s'',a')$. Now consider the labeling of (s,s') in $M \parallel M'$. This state must have labeling $L'(s',a')$ on the state component a'. Hence $((s,s'),s'')$ could not be a state of M_0, a contradiction.

We now know that (s',s'') is a state of $M' \parallel M''$. If $(s,(s',s''))$ is not in S_1, then there is some state component a such that the labeling of s and the labeling of (s',s'') disagree on this label. This state component a must be in one of A' or A''; let us suppose it is in A'. The labeling of (s',s'') on a is then the same as the labeling of s' on a. As a result, we have $L(s,a) \neq L'(s',a)$, and hence (s,s') is not a state of $M \parallel M'$. This means that $((s,s'),s'')$ is not a state of M_0, a contradiction. Similarly, we obtain a contradiction if a is in A'' instead of A'. Thus, we conclude that $(s,(s',s''))$ must indeed be a state of M_1.

Now that we know ϕ is a bijection between the states of M_0 and M_1, it is easy to see that initial states, transitions, labelings, and acceptance conditions are preserved. Thus, ϕ is in fact an isomorphism between the structures. □

We also prove that the composition operation on structures corresponds to the composition operation on Moore machines (proposition 3.1).

Proof We are given composable Moore machines M and M', and we want to know that *struct*$(M \parallel M')$ is isomorphic to *struct*$(M) \parallel$ *struct*(M'). Let $M'' = M \parallel M'$, and define ϕ by:

$$\phi(((s,s'),f'')) = ((s, f'' \cup (L'(s') \downarrow A_I)), (s', f'' \cup (L(s) \downarrow A'_I))).$$

First note that $(s, f'' \cup (L'(s') \downarrow A_I))$ is a state of *struct*(M), and that its labeling is compatible with the state $(s', f'' \cup (L(s) \downarrow A'_I))$ of *struct*(M'); hence ϕ is a well-defined mapping between the two sets of states. It is clearly an injection. Given a state $((s,f),(s',f'))$ of the composition of the structures, if we let $f'' = f \downarrow A''_I = f' \downarrow A''_I$, then we obtain a state of *struct*(M'') mapping to $((s,f),(s',f'))$; hence ϕ is a bijection.

ϕ obviously preserves labelings and initial states, and the acceptance conditions of both structures are empty. If $((s_0,s'_0),f''_0)$ can transition to $((s_1,s'_1),f''_1)$ in *struct*(M''), then $R''((s_0,s'_0),f''_0,(s_1,s'_1))$. This implies $R(s_0, f''_0 \cup (L'(s'_0) \downarrow A_I), s_1)$ and $R'(s'_0, f''_0 \cup (L(s_0) \downarrow A'_I), s'_1)$. Now $(s_0, f''_0 \cup (L'(s'_0) \downarrow A_I))$ is a state of *struct*(M) that can transition to $(s_1, f''_1 \cup (L'(s'_1) \downarrow A_I))$, and similarly $(s'_0, f''_0 \cup (L(s_0) \downarrow A'_I))$ can transition to $(s'_1, f''_1 \cup (L(s_1) \downarrow A'_I))$. Hence $\phi(((s_0,s'_0),f''_0))$ can transition to $\phi(((s_1,s'_1),f''_1))$. A similar argument shows that transitions in *struct*$(M) \parallel$ *struct*(M') are also transitions in *struct*(M''). Thus, ϕ is an isomorphism. □

We next return to the proofs of theorems 3.1 through 3.4. Recall that the first of these states that $\preceq$ is the largest simulation relation under the set inclusion ordering.

Proof First, we show that $\preceq$ is in fact a simulation relation. Suppose $s \preceq s'$. Hence, there exists some $\sqsubseteq$ that is a simulation relation and for which $s \sqsubseteq s'$. Since $\sqsubseteq$ is a simulation relation, $L(s) \downarrow A' = L'(s')$. Let $\pi = s_0 s_1 s_2 \ldots$ be a path in M starting at $s = s_0$. Again, since $\sqsubseteq$ is a simulation relation, there exists a path $\pi' = s'_0 s'_1 s'_2 \ldots$ starting at $s' = s'_0$ such that for all i, $s_i \sqsubseteq s'_i$. Since there exists a simulation relation ($\sqsubseteq$) relating each s_i and s'_i, $s_i \preceq s'_i$. Hence $\preceq$ satisfies the conditions for a simulation relation.

If $\sqsubseteq$ is any simulation relation for which $s \sqsubseteq s'$, then by definition $s \preceq s'$. Hence $\sqsubseteq \subseteq \preceq$. □

To show that $\preceq$ is a preorder (theorem 3.2), we just argue that it is transitive (reflexivity is obvious).

Proof Suppose $M \preceq M'$ and $M' \preceq M''$. Obviously, the condition $A \subseteq A''$ holds. Define $\sqsubseteq$ as the relational product of the two simulation relations:

$$\sqsubseteq = \{\, (s, s'') \mid \exists s' [s \preceq s' \wedge s' \preceq s''] \,\}.$$

We first show that $\sqsubseteq$ is a simulation relation.

Suppose $s \sqsubseteq s''$. Let s' be a state such that $s \preceq s'$ and $s' \preceq s''$. Since $\preceq$ is a simulation relation, $L'(s') \downarrow A'' = L''(s'')$. Similarly, $L(s) \downarrow A' = L'(s')$, so $L(s) \downarrow A'' = L''(s'')$. Let $\pi = s_0 s_1 s_2 \ldots$ be a path in M starting at $s = s_0$. There must exist a path $\pi' = s'_0 s'_1 s'_2 \ldots$ starting at $s' = s'_0$ such that for all i, $s_i \preceq s'_i$. For this path, there must exist a path $\pi'' = s''_0 s''_1 s''_2 \ldots$ starting at $s'' = s''_0$ such that for all i, $s'_i \preceq s''_i$. By definition, $s_i \sqsubseteq s''_i$ for all i, i.e., π and π'' are paths from s and s'' related by $\sqsubseteq$. Thus, $\sqsubseteq$ is a simulation relation.

Now, if we can show that every initial state of M is related to a corresponding initial state of M'' by $\sqsubseteq$, then we are done. Let $s \in I$. Since $M \preceq M'$, there is some $s' \in I'$ such that $s \preceq s'$. Similarly, since $M' \preceq M''$, there is $s'' \in I''$ such that $s' \preceq s''$. By definition, $s \sqsubseteq s''$. □

To prove that $\|$ respects $\preceq$, we first prove the following lemma. It tells us that paths in a composition correspond to paths in the components.

Lemma 3.1 (path lemma) Let $M'' = M \parallel M'$. The following conditions are equivalent.

1. $\pi'' = (s_0, s'_0)(s_1, s'_1)(s_2, s'_2) \ldots$ is a path in M''.

2. $\pi = s_0 s_1 \ldots$ and $\pi' = s'_0 s'_1 \ldots$ are paths in M and M' respectively, and (s_i, s'_i) is a state of M'' for all i.

Proof If $\pi'' = (s_0, s'_0)(s_1, s'_1)(s_2, s'_2) \ldots$ is a path in M'', then obviously (s_i, s'_i) is a state of M'' for each i. By definition of composition, we must also have $R(s_i, s_{i+1})$ and $R'(s'_i, s'_{i+1})$. Finally, suppose that (P, Q) is a pair in the acceptance condition of M. Then $((P \times S') \cap S'', (Q \times S') \cap S'')$ is a pair in the acceptance condition of M''. Since π'' is a path, either there is some i such that $(s_j, s'_j) \in (P \times S') \cap S''$ for all $j \geq i$, or there are infinitely many i such that $(s_i, s'_i) \in (Q \times S') \cap S''$. This implies that either almost all of the s_i are in P, or infinitely many of them are in Q. Thus, π is a path, and a similar argument shows that π' is.

Conversely, if π and π' are paths in M and M' respectively, then we must have $R(s_i, s_{i+1})$ and $R'(s'_i, s'_{i+1})$. By definition of composition and the fact that (s_i, s'_i) is a state of M'' for all i, we obtain $R''((s_i, s'_i), (s_{i+1}, s'_{i+1}))$ for all i. Let $((P \times S') \cap S'', (Q \times S') \cap S'')$ be one of the pairs in the acceptance condition of M'' (assume without loss of generality that it derives from $(P, Q) \in F$). Since π is a path in M, either infinitely many s_i are in Q, or almost all of them are in P. This implies that either infinitely many (s_i, s'_i) are in $(Q \times S') \cap S''$, or almost all of them are in $(P \times S') \cap S''$. Thus, each pair in the acceptance condition of M'' is satisfied, so π'' is a path of M''. □

We now prove that composition respects simulation.

Proof Assume $M \preceq M'$. Let $M_0 = M \parallel M''$ and $M_1 = M' \parallel M''$. Define $\sqsubseteq$ to be the relation

$$\{ ((s, s''), (s', s'')) \mid (s, s'') \in S_0 \wedge (s', s'') \in S_1 \wedge s \preceq s' \}.$$

Suppose $(s, s'') \sqsubseteq (s', s'')$. Let a be a state component of M_1. If $a \in A'$, then $L_0((s, s''), a) = L(s, a)$ (since $A \supseteq A'$). But $L_1((s', s''), a) = L'(s', a) = L(s, a)$ since $s \preceq s'$. If $a \in A''$, then $L_0((s, s''), a) = L''(s'', a)$, and $L_1((s', s''), a) = L''(s'', a)$ as well. In both cases, the state labelings agree on a. Now let $\pi_0 = (s_0, s''_0)(s_1, s''_1)(s_2, s''_2)\ldots$ be a path in M_0 from $(s, s'') = (s_0, s''_0)$. By the path lemma, we can project this to paths π from s in M and π'' from s'' in M''. Since $s \preceq s'$, there is a path $\pi' = s'_0 s'_1 s'_2 \ldots$ from $s' = s_0$ in M' such that $s_i \preceq s'_i$ for each i. Again by the path lemma, the paths π' and π'' can be combined into a path $\pi_1 = (s'_0, s''_0)(s'_1, s''_1)(s'_2, s''_2)\ldots$ in M_1. By definition, corresponding states on π_0 and π_1 are related by $\sqsubseteq$, and hence $\sqsubseteq$ is a simulation relation.

If $(s, s'') \in I_0$, then $s \in I$ and $s'' \in I''$. Since $M \preceq M'$, there is some s' such that $s \preceq s'$ and $s' \in I'$. For this s', $(s, s'') \sqsubseteq (s', s'')$. Hence $(s, s'') \preceq (s', s'')$, and every initial state of M_0 has a corresponding initial state of M_1. Thus, $M_0 \preceq M_1$. □

The proof that $M \preceq M \parallel M$ (theorem 3.4) is straightforward; we just observe that $\{ (s,(s,s)) \mid s \in S \}$ is a simulation relation. We now consider theorem 3.5 (the observation that composing M with $\top(B)$ for $B \subseteq A$ leads to a structure isomorphic to M).

Proof Let $s \in S$. The only state of $\top(B)$ that has a compatible labeling with s is $L(s) \downarrow B$. Hence the mapping ϕ defined by $\phi(s) = (s, L(s) \downarrow B)$ is a bijection between states of M and states of $M \parallel \top(B)$. Clearly ϕ preserves labelings. Since the all states of $\top(B)$ are initial and all pairs of states have a transition between them, ϕ also preserves initial states and transitions. Similarly, F is mapped to a corresponding acceptance condition in the composition. □

The remaining task is to prove the correctness of the tableau construction. Earlier we mentioned that the construction as given earlier does not handle certain degenerate cases, so we first discuss these cases and the changes that need to be made. Consider the formula $\mathbf{A}(\mathit{false}\, \mathbf{V}\, \mathit{false})$ ($\mathbf{AG}\, \mathit{false}$). If M is a structure where there is no initial state that is the start of a path, then M actually satisfies this formula. However, if we construct the tableau according to the earlier definition, we find that it has no initial states. This is because $\Phi(\mathbf{A}(\mathit{false}\ \mathbf{V}\ \mathit{false})) = \Phi(\mathit{false}) \cap \ldots$, and $\Phi(\mathit{false})$ is the empty set. Since the tableau has no initial states, it may not be able to simulate M. The solution to handling cases such as this is to recognize that formulas such as $\mathbf{A}(\psi\ \mathbf{U}\ \chi)$ and $\mathbf{A}(\psi\ \mathbf{V}\ \chi)$ will be true for states that are the start of no path, regardless of ψ and χ. To take this into account, we extend the set of elementary formulas. When φ has a subformula involving $\mathbf{U}$ or $\mathbf{V}$, we add a special formula $\mathbf{AX}\, \mathit{false}$ to the elementary formulas of φ. Then, we alter the mapping Φ so that $\Phi(\mathbf{A}(\psi\ \mathbf{V}\ \chi))$ is

$$(\Phi(\chi) \cap (\Phi(\psi) \cup \Phi(\mathbf{AX}\ \mathbf{A}(\psi\ \mathbf{V}\ \chi))) \cup \Phi(\mathbf{AX}\, \mathit{false})$$

and $\Phi(\mathbf{A}(\psi\ \mathbf{U}\ \chi))$ is

$$(\Phi(\chi) \cup (\Phi(\psi) \cap \Phi(\mathbf{AX}\ \mathbf{A}(\psi\ \mathbf{U}\ \chi))) \cup \Phi(\mathbf{AX}\, \mathit{false}).$$

With these changes, the construction is correct in all cases, as we now show.

First, we demonstrate that if $M \preceq \mathcal{T}(\varphi)$, then $M \models \varphi$. This will be done in two steps:

1. Prove that if $M \preceq M'$ and if M' satisfies a formula, then M satisfies the formula as well.

2. Prove that $\mathcal{T}(\varphi) \models \varphi$.

Then we will have $M \preceq \mathcal{T}(\varphi)$, $\mathcal{T}(\varphi) \models \varphi$, and hence $M \models \varphi$.

Lemma 3.2 If $M \preceq M'$ and φ is a formula with *comp*$(\varphi) \subseteq A'$, then $M' \models \varphi$ implies $M \models \varphi$.

Proof The proof of this theorem is very similar in spirit to that of theorem 2.2. Clearly it is enough to show that if $s' \models \varphi$ and $s \preceq s'$, then $s \models \varphi$. We proceed by induction on the structure of formulas.

1. For atomic formulas and their negations, the result is obvious. For conjunctions and disjunctions, it follows immediately from the induction hypothesis.

2. Consider a formula of the form $\mathbf{A}(\varphi \, \mathbf{U} \, \psi)$. Let $\pi = s_0 s_1 s_2 \ldots$ be a path from $s = s_0$; we want to show that this path satisfies $\varphi \, \mathbf{U} \, \psi$. Since $s \preceq s'$, there is a path $\pi' = s'_0 s'_1 s'_2 \ldots$ from $s' = s'_0$ that corresponds to π. For each i, $s_i \preceq s'_i$. Hence by the induction hypothesis, $s'_i \models \varphi$ implies $s_i \models \varphi$, and similarly for ψ. If π does not satisfy $\varphi \, \mathbf{U} \, \psi$, then this implies that π' does not satisfy $\varphi \, \mathbf{U} \, \psi$ either. Hence $s' \not\models \mathbf{A}(\varphi \, \mathbf{U} \, \psi)$, a contradiction. Thus we conclude that $s \models \mathbf{A}(\varphi \, \mathbf{U} \, \psi)$.

3. The cases for $\mathbf{AX}\,\varphi$ and $\mathbf{A}(\varphi \, \mathbf{V} \, \psi)$ are similar to the above. □

Lemma 3.3 Let s be a state of $\mathcal{T}(\varphi)$. For all subformulas ψ of φ, if $s \in \Phi(\psi)$, then $s \models \psi$. Hence $\mathcal{T}(\varphi) \models \varphi$.

Proof Let $M = \mathcal{T}(\varphi)$ and $s = (f, E)$; we proceed by induction on the structure of the subformula.

1. For *true*, we have that $\Phi(\mathit{true})$ contains every state, so $s \in \Phi(\mathit{true})$ iff $s \models \mathit{true}$.

2. For a subformula of the form $a = d$, we have
$$\Phi(a = d) = \{ (f, E) \mid f(a) = d \}.$$
$s \models a = d$ iff $L(s, a) = d$, and from the definition of $\mathcal{T}(\varphi)$, we have $L((f, E), a) = f(a)$. Hence $s \in \Phi(a = d)$ iff $s \models a = d$.

3. For the negation of an atomic formula, just note that the above two cases are iff, and that $\Phi(\neg\psi) = S - \Phi(s)$.

4. $\Phi(\psi \wedge \chi) = \Phi(\psi) \cap \Phi(\chi)$. If $s \in \Phi(\psi \wedge \chi)$, then by the induction hypothesis, $s \models \psi$ and $s \models \chi$. Hence $s \models \psi \wedge \chi$. Similarly, if $s \in \Phi(\psi \vee \chi)$, then $s \models \psi \vee \chi$.

5. For subformulas of the form $\mathbf{AX}\,\psi$, we have $s \in \Phi(\mathbf{AX}\,\psi)$ iff $\mathbf{AX}\,\psi \in E$. In other words, Φ gives exactly those states labeled with the elementary subformula $\mathbf{AX}\,\psi$. Suppose $s = s_0 \in \Phi(\mathbf{AX}\,\psi)$. By definition of the tableau, if $R(s_0, s_1)$, then $s_1 \in \Phi(\psi)$. Applying the induction hypothesis, we find $s_1 \models \psi$. Since every successor of s_0 must satisfy ψ, $s_0 \models \mathbf{AX}\,\psi$.

   For a subformula of the form $\mathbf{A}(\psi\ \mathbf{V}\ \chi)$, $\Phi(\mathbf{A}(\psi\ \mathbf{V}\ \chi))$ is
$$(\Phi(\chi) \cap (\Phi(\psi) \cup \Phi(\mathbf{AX}\,\mathbf{A}(\psi\ \mathbf{V}\ \chi)))) \cup \Phi(\mathbf{AX}\,\mathit{false}).$$
   If $s \in \Phi(\mathbf{AX}\,\mathit{false})$, then there are no paths starting at s, so it satisfies $\mathbf{A}(\psi\ \mathbf{V}\ \chi)$. Otherwise, $s \in \Phi(\chi)$, so by the induction hypothesis, $s \models \chi$. Also, $s \in \Phi(\psi) \cup \Phi(\mathbf{AX}\,\mathbf{A}(\psi\ \mathbf{V}\ \chi))$. If $s \in \Phi(\psi)$, then $s \models \psi$ by the induction hypothesis. If instead, $s = s_0 \in \Phi(\mathbf{AX}\,\mathbf{A}(\psi \mathbf{V} \chi))$, then by definitions of Φ and R, if $R(s_0, s_1)$, then $s_1 \in \Phi(\mathbf{A}(\psi \mathbf{V} \chi))$. Thus, in this case, all successors of s must also be in $\Phi(\mathbf{A}(\psi\ \mathbf{V}\ \chi))$. Let $\pi = s_0 s_1 s_2 \ldots$ be a path starting at $s = s_0$. Suppose $s_i \not\models \psi$ for all $i < j$. By the above, we must have $s_j \models \chi$. Hence $\psi\,\mathbf{V}\,\chi$ is true along the path, and since π was arbitrary, $s \models \mathbf{A}(\psi\ \mathbf{V}\ \chi)$.

   The argument for subformulas of the form $\mathbf{A}(\psi\ \mathbf{U}\ \chi)$ is similar to that for $\mathbf{A}(\psi\ \mathbf{V}\ \chi)$.

Now if s is an initial state of the tableau, then by definition $s \in \Phi(\varphi)$. Hence $s \models \varphi$, and so $\mathcal{T}(\varphi) \models \varphi$. □

This concludes one direction of the proof. Now we want to prove that if $M' \models \varphi$, then $M' \preceq \mathcal{T}(\varphi)$. This will be done by constructing an explicit simulation relation between M' and $\mathcal{T}(\varphi)$. The idea will be to take a state s' of M', look at its labeling and the elementary formulas that it satisfies, and use this to construct a unique state of $\mathcal{T}(\varphi)$ that can simulate s'. First, we define what will be the simulation relation and prove a sort of analog to the converse of lemma 3.3.

Lemma 3.4 Let $M = \mathcal{T}(\varphi)$, and let M' be a structure with $A' \supseteq A$. Define $\sqsubseteq$ on $S' \times S$ by $s' \sqsubseteq (f, E)$ iff the following conditions hold:

1. $L'(s') \downarrow A = f$.

2. For every $\mathbf{AX}\,\psi \in el(\varphi)$, $\mathbf{AX}\,\psi \in E$ iff $s' \models \mathbf{AX}\,\psi$.

Then $s' \sqsubseteq s$ implies that for every subformula or elementary formula ψ of φ, $s' \models \psi$ implies $s \in \Phi(\psi)$.

Proof By induction on the structure of formulas. In this proof, the base cases are the atomic subformulas and the elementary subformulas. In all cases, assume $s' \sqsubseteq s = (f, E)$.

1. ᒥ͏r *true*, $s' \models true$ iff $s \in \Phi(true)$.

2. For a subformula $a = d$, we get that $s' \models a = d$ iff $L'(s', a) = d$ iff $L((f, E), a) = d$ iff $f(a) = d$ iff $(f, E) \in \Phi(a = d)$.

3. For a negated atomic subformula, the result follows from the facts that $\Phi(\neg\psi) = S - \Phi(\psi)$ and that the above two cases are iffs.

4. If s' satisfies an elementary formula $\mathbf{AX}\,\psi$, then by definition of $\sqsubseteq$, $\mathbf{AX}\,\psi \in E$. But $(f, E) \in \Phi(\mathbf{AX}\,\psi)$ iff $\mathbf{AX}\,\psi \in E$.

5. For a subformula such as $\psi \wedge \chi$, we get that s' must satisfy ψ and χ. By the induction hypothesis, $s \in \Phi(\psi)$ and $s \in \Phi(\chi)$. Hence $s \in \Phi(\psi) \cap \Phi(\chi)$, and $s \in \Phi(\psi \wedge \chi)$. Subformulas of the form $\varphi \vee \psi$ are handled in a similar mannei.

6. Subformulas of the form $\mathbf{AX}\,\psi$ are elementary formulas and were dealt with above. Consider a subformula of the form $\mathbf{A}(\psi\,\mathbf{V}\,\chi)$. If s' is not the start of some path, then $s' \models \mathbf{AX}\,\mathit{false}$, so $\mathbf{AX}\,\mathit{false} \in E$, and hence $s \in \Phi(\mathbf{A}(\psi\,\mathbf{V}\,\chi))$. Assume s' is the start of a path. If $s' \models \mathbf{A}(\psi\,\mathbf{V}\,\chi)$, then we first have that $s' \models \chi$. By the induction hypothesis, $s \in \Phi(\chi)$. Also, either $s' \models \psi$, or every successor of s' must satisfy $\mathbf{A}(\psi\,\mathbf{V}\,\chi)$. In the former case, $s \in \Phi(\psi)$. In the latter, s' must satisfy $\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{V}\,\chi)$. This is an elementary subformula, and hence by the induction hypothesis, $s \in \Phi(\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{V}\,\chi))$. From these two cases we can conclude $s \in \Phi(\psi) \cup \Phi(\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{V}\,\chi))$. All together, we have

$$s \in \Phi(\chi) \cap (\Phi(\psi) \cup \Phi(\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{V}\,\chi))),$$

and so $s \in \Phi(\mathbf{A}(\psi\,\mathbf{V}\,\chi))$.

Consider a subformula of the form $\mathbf{A}(\psi\,\mathbf{U}\,\chi)$. If s' is not the start of some path, then as above we have $s' \in \Phi(\mathbf{A}(\psi\,\mathbf{U}\,\chi))$. Otherwise, either $s' \models \chi$, or $s' \models \psi$ and every successor of s' satisfies $\mathbf{A}(\psi\mathbf{U}\chi)$. In the latter case $s' \models \mathbf{AX}\,\mathbf{A}(\psi\mathbf{U}\chi)$. Applying the induction hypothesis, we find

$$s \in \Phi(\chi) \cup (\Phi(\psi) \cap \Phi(\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi))).$$

Hence $s \in \Phi(\mathbf{A}(\psi\,\mathbf{U}\,\chi))$. □

Now, using this result, we have:

Lemma 3.5 The relation $\sqsubseteq$ given above is a simulation relation.

Proof Assume $s' \sqsubseteq s = (f, E)$. By definition, $L'(s') \downarrow A = f = L(s)$. Suppose now that $\pi' = s'_0 s'_1 s'_2 \ldots$ is a path from $s' = s'_0$ in M'. We will construct a path π from $s = s_0$ in M that corresponds to π'. Assume that we have constructed states up to s_i so far, and that we know $s'_i \sqsubseteq s_i$. Let $\mathbf{AX}\,\psi_0, \ldots, \mathbf{AX}\,\psi_{m-1}$ be the elementary formulas that s'_i satisfies. Then s'_{i+1} must satisfy $\psi_0, \ldots, \psi_{m-1}$. Now observe that each state of M' is related to a (unique) state of M by $\sqsubseteq$. Let s_{i+1} be the state related to s'_{i+1} in this manner. By the previous lemma, $s_{i+1} \in \Phi(\psi_0), \ldots, s_{i+1} \in \Phi(\psi_{m-1})$. Since $s'_i \sqsubseteq s_i = (f_i, E_i)$, we know that the elementary formulas $\mathbf{AX}\,\psi_j$ are the only elementary formulas for which $\mathbf{AX}\,\psi_j \in E_i$. Then by the definition of $\mathcal{T}(\varphi)$, $R(s_i, s_{i+1})$. Thus, we have found s_{i+1} that extends the sequence and for which $s'_{i+1} \sqsubseteq s_{i+1}$. Now we just have to show that this sequence satisfies the acceptance conditions.

Assume that it does not. Then, looking at the acceptance conditions for the tableau, we see that there must be some elementary formula $\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi)$ and some i such that for all $j \geq i$:

$$s_j \notin (S - \Phi(\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi))) \cup \Phi(\chi).$$

Then $s_j = (f_j, E_j)$ is not in either part of the union. Now $s_j \notin S - \Phi(\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi))$ implies that $\mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi) \in E_j$. By the definition of $\sqsubseteq$, we find that $s'_j \models \mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi)$. Further, since $s_j \notin \Phi(\chi)$, then by the previous lemma, we must have $s'_j \not\models \chi$. But then we have $s'_i \models \mathbf{AX}\,\mathbf{A}(\psi\,\mathbf{U}\,\chi)$, and for all $j \geq i$, $s'_j \not\models \chi$. This implies that π' must not be a path, a contradiction. □

Putting the previous two lemmas together, we obtain the desired result. If $M' \models \varphi$, then by definition, every initial state s' of M' satisfies φ. Recall the simulation relation $\sqsubseteq$ defined above pairs every such s' with a unique state s of the tableau. Now lemma 3.4 implies that s is in $\Phi(\varphi)$, and hence by the definition of the tableau, s is an initial state. Since $\sqsubseteq$ is a simulation relation, we conclude that s' can be simulated by an initial state of the tableau. Hence $M \preceq \mathcal{T}(\varphi)$.

Chapter 4

Abstraction

So far, all of the methods we have for checking $\preceq$ (and $\models$) are either direct algorithms (e.g., model checking) or techniques based on properties of $\preceq$ and $\|$ (e.g., assume-guarantee proofs). In this section, we consider methods based on ***abstraction***. When performing abstractions, we lose information about the exact behavior of the system under consideration. As a result, there will be some properties whose truth cannot be determined by looking only at the abstracted system. It is important that the verification methodology not lead to false positive results. That is, if we find that some property is true of the abstracted system, we must have a guarantee that the property really does hold for the actual system. A verification methodology with this property is said to be *conservative*. Note that we have no requirements about what happens in the actual system if the property is not true in the abstracted system.

Our main goal is to be able to verify efficiently systems that manipulate data in nontrivial ways. For such systems, we will want to collapse the possible data values down to a small set of abstract elements. There are two main reasons why abstraction is useful for verifying systems that manipulate data. First, the properties that we are interested in proving can often be expressed in terms of abstract values, i.e., we can write accurate specifications at the abstract level. Second, real systems generally manipulate data in well-structured ways. As a result, we can tell something about the abstract value representing the result of an operation based on the abstract values of the inputs. This is important if we are going to make a model of our system that is not too conservative.

4.1 Conservative Connections

When using abstraction for verification, we will be working at two levels: a concrete one and an abstract one. Structures at the abstract level will be viewed as approximations to structures at the concrete level. In order to tie the levels together, we introduce a map that takes a structure at the concrete level and produces an abstract-level view of it. Another map will take a structure at the abstract level and give us a concrete-level structure that represents the "most general" behavior corresponding to the abstract structure. The goal of using abstraction is to check a specification at the abstract level, and then to infer a similar relationship at the concrete level. Thus, we are led to the following definition.

Definition 4.1 Let Ψ_u be a function mapping structures over A to structures over $\hat{A}$, and let Ψ_l be a function mapping structures over $\hat{A}$ to structures over A. We say that (Ψ_u, Ψ_l) is a conservative connection (between structures over A and $\hat{A}$) when for all structures M and $\widehat{M}$ (over A and $\hat{A}$ respectively), $\Psi_u(M) \preceq \widehat{M}$ implies $M \preceq \Psi_l(\widehat{M})$.

The notion above is a kind of hybrid of the conservative approximation of Burch [21] and the Galois connections used by Bensalem *et al.* [6], and also has some relation to Kurshan's automata homomorphisms [62]. (Actually, we can impose a lattice structure on structures: meet is composition, join is a kind of disjoint union, and top and bottom are the structures $\top$ and $\bot$ of example 3.8. Then the definition above can actually be viewed as a Galois connection between the lattice of structures over A and the lattice of structures over $\hat{A}$.)

The motivation behind using a conservative connection is that verifying $\Psi_u(M) \preceq \widehat{M}$ will be easier than verifying $M \preceq \Psi_l(\widehat{M})$ directly. The price we pay for the simplification is that we may obtain false negative results: it may be that $M \preceq \Psi_l(\widehat{M})$ while $\Psi_u(M) \not\preceq \widehat{M}$. The condition in the definition of a conservative connection can be expressed pictorially as in figure 4.1.

$$\begin{tikzcd}
\Psi_u(M) \arrow[r, dash] & \preceq \arrow[r, dash] \arrow[d, dotted, "\text{implies}"] & \widehat{M} \arrow[d, "\Psi_l"] \\
M \arrow[u, "\Psi_u"] \arrow[r, dash] & \preceq \arrow[r, dash] & \Psi_l(\widehat{M})
\end{tikzcd}$$

Figure 4.1: A conservative connection

Note that, in contrast to the conservative approximations of Burch, the mapping Ψ_l has abstract structures as its domain rather than its range. This is for two reasons: first, it is often most convenient to give the specification at an abstract level; Ψ_l tells what the specification means at the level of the implementation. Second, in our framework it will generally mathematically cleaner to give a single "most general" structure represented by a specification than to give the specification corresponding to an arbitrary structure. An implementation might actually provide a kind of lower bound mapping from structures over A to structures over $\hat{A}$. Applying this mapping to M and then applying Ψ_l to the result should give something smaller than M under $\preceq$. Also, an implementation may not actually provide a way to compute Ψ_u; if instead it produces something larger (under $\preceq$), this is still sufficient for verification purposes.

Example 4.1 Let $A = \hat{A}$, and let *id* be the identity mapping between structures over A. Then (id, id) is a conservative connection. □

Recall that we can have $M \preceq M'$ when $A' \subseteq A$. We could have actually defined $\preceq$ so that it only held between structures with the same sets of visible state components. Then we would use a conservative connection to hide state components. As another example of a conservative connection, we now consider how this would be done. First though, we will need a hiding operation for structures. We choose one that is analogous to our hiding operation for Moore machines (definition 2.14).

Definition 4.2 Let M be a structure and $\hat{A}$ be a set of state components. The result of ***restricting*** *M to $\hat{A}$* (denoted $M \downarrow \hat{A}$) is the structure $\widehat{M}$ defined by:

1. $\hat{S} = S$.
2. $\hat{I} = I$.
3. $\hat{R} = R$.
4. $\hat{L}$ is defined by $\hat{L}(s) = L(s) \downarrow \hat{A}$.
5. $\hat{F} = F$.

Example 4.2 Let $\hat{A} \subseteq A$, and let $\Psi_u(M) = M \downarrow \hat{A}$. Also, take $\Psi_l(\widehat{M}) = \widehat{M} \parallel \mathsf{T}(A - \hat{A})$. Then (Ψ_u, Ψ_l) is a conservative connection. To see this, assume $\Psi_u(M) \preceq \widehat{M}$. We note that $M \preceq \Psi_u(M)$, so $M \preceq \widehat{M}$. Composing both sides with $\mathsf{T}(A - \hat{A})$ gives

$$M \parallel \mathsf{T}(A - \hat{A}) \preceq \widehat{M} \parallel \mathsf{T}(A - \hat{A}).$$

Now by theorem 3.5, $M \parallel \mathsf{T}(A - \hat{A})$ is isomorphic to M. Also, the right side of the above relation is $\Psi_l(\widehat{M})$, so we have $M \preceq \Psi_l(\widehat{M})$, as required. □

Example 4.3 The composition of conservative connections is also a conservative connection. Suppose that (Ψ_u, Ψ_l) is a conservative connection between structures over A and $\hat{A}$ and (Ψ'_u, Ψ'_l) is a conservative connection between structures over $\hat{A}$ and $\tilde{A}$. Then $(\Psi'_u \circ \Psi_u, \Psi_l \circ \Psi'_l)$ is a conservative connection between A and $\tilde{A}$. □

Example 4.4 Suppose (Ψ_u, Ψ_l) is a conservative connection between structures over A and $\hat{A}$. If Ψ'_u and Ψ'_l are functions with $\Psi'_u(M) \preceq \Psi_u(M)$ and $\Psi_l(\widehat{M}) \preceq \Psi'_l(\widehat{M})$, then (Ψ'_u, Ψ'_l) is also a conservative connection. □

The mappings in conservative connections often have other nice properties. First, they are commonly ***monotonic*** with respect to the preorder $\preceq$. So, for example, applying Ψ_u to M and to M' with $M \preceq M'$ gives $\Psi_u(M) \preceq \Psi_u(M')$. Second, distributing the mapping Ψ_u over a composition gives something larger under $\preceq$. This latter property is especially important: in order to use conservative connections effectively, we usually do not want to deal explicitly with M when producing the abstract version of M. The property says that we can approximate the parts of a composition before composing and still remain conservative.

Example 4.5 Recall the earlier example of collapsing states with identical labelings (example 3.10). (*collapse*, *collapse*) is a conservative connection between structures over A and A. Also, *collapse* is monotonic, and it can be distributed over compositions (we prove this later).

Note that applying *collapse* to a structure in which each state has a unique labeling function gives a structure isomorphic to the original one. When $\Psi_l(\Psi_u(M)) \preceq M$, then we say that the conservative connection M is *exact* for M. Thus, (*collapse*, *collapse*) is exact for structures in which each state has a unique labeling function. □

We now consider conservative approximations that abstract the visible state components of a structure. The abstraction will be given in terms of a set of mappings on state component values. That is, for each concrete state component a, we will have a corresponding abstract state component $\hat{a}$. Then we will provide a mapping between D_a and $D_{\hat{a}}$ that will be used to give an abstract-level view of the value of a. If we simply apply this mapping to the state labelings of a concrete-level structure, that will give us the desired abstract-level structure. This is the analog of an automata homomorphism induced by a boolean algebra homomorphism in the work of Kurshan [62].

Definition 4.3 Let $A = \{a_0, \ldots, a_{n-1}\}$, $\hat{A} = \{\widehat{a_0}, \ldots, \widehat{a_{n-1}}\}$, and suppose $h_0, \ldots, h_{n-1}$ are surjections with $h_i \colon D_{a_i} \rightarrow D_{\hat{a}_i}$. Let h be the function mapping labeling functions over A to labeling functions over $\hat{A}$ defined by

$$(h(f))(\hat{a}_i) = h_i(f(a_i)).$$

Let M be a structure over A. Define $abs_u(M)$ *(with respect to h)* to be the following structure $\widehat{M}$:

1. $\hat{S} = S$.
2. $\hat{I} = I$.
3. $\hat{R} = R$.
4. $\hat{L}(s) = h(L(s))$.
5. $\hat{F} = F$.

This gives us a mapping from concrete-level structures to abstract-level structures. Now we want to produce a conservative connection, and so far we have the situation shown in figure 4.2. We need to define abs_l taking us from the abstract level to the concrete level.

$$\begin{tikzcd}
abs_u(M) \arrow[r, dash] & \preceq \arrow[r, dash] \arrow[d, dotted, "\text{implies}"] & \widehat{M} \arrow[d, "?"] \\
M \arrow[u, "abs_u"] \arrow[r, dash] & \preceq \arrow[r, dash] & ?
\end{tikzcd}$$

Figure 4.2: Situation after defining abs_u

Suppose that M is a concrete-level structure and that (s_0, s_1) is a transition of M. Also suppose that there is one visible state component a that can take on the values $\{0, 1, 2, 3\}$. We will assume that the labeling for s_0 has $a = 0$ and the labeling for s_1 has $a = 1$. Let h map 0 and 2 to *even* and map 1 and 3 to *odd*. When we apply abs_u to M, we get states s_0 and s_1 with labelings *even* and *odd*, respectively, and a transition between them. Now suppose that we have a structure $\widehat{M}$ that can simulate $abs_u(M)$. There should be some transition $(\widehat{s_0}, \widehat{s_1})$ that can simulate the (s_0, s_1) transition of $abs_u(M)$. This implies that $\hat{L}(\widehat{s_0})$ should be *even* and $\hat{L}(\widehat{s_1})$ should be *odd*. Now we want to define a mapping abs_l from abstract to concrete structures. Because simulation at the abstract level should imply simulation at the concrete level, we will want $abs_l(\widehat{M})$ to be able to simulate M. It is natural to use the $(\widehat{s_0}, \widehat{s_1})$ transition to construct a transition of $abs_l(\widehat{M})$ that can simulate the (s_0, s_1) transition of M. However, since h is generally not a bijection, given just the labelings of $\widehat{s_0}$ and $\widehat{s_1}$, we cannot tell exactly what labelings s_0 and s_1 have. Thus, we will expand each state of $\widehat{M}$ into a class of sta·es, one for each compatible labeling. This will give us the state space of $abs_l(\widehat{M})$. In this example, we expand $\widehat{s_0}$ into two states, $(\widehat{s_0}, 0)$ and $(\widehat{s_0}, 2)$ (where 0 and 2 denote the labeling functions mapping a to 0 and 2). Similarly, $\widehat{s_1}$ expands into $(\widehat{s_1}, 1)$ and $(\widehat{s_1}, 3)$. Since s_0 has the labeling $a = 0$, we choose $(\widehat{s_0}, 0)$ to simulate it, and likewise $(\widehat{s_1}, 1)$ will simulate s_1. Now we just include all transitions from states in the class for $\widehat{s_0}$ to states in the class for $\widehat{s_1}$. The $((\widehat{s_0}, 0), (\widehat{s_1}, 1))$ transition will simulate the (s_0, s_1) transition of M. We now define abs_l formally.

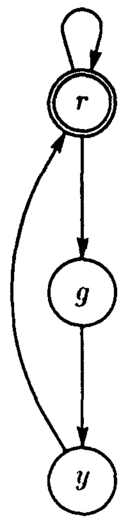
Definition 4.4 Let $\widehat{M}$ be a structure over $\hat{A}$. Define $abs_l(\widehat{M})$ *(with respect to h)* to be the structure M given by:

1. $S = \{ (\hat{s}, f) \mid \hat{s} \in \hat{S} \wedge f \in labelings(A) \wedge \hat{L}(\hat{s}) = h(f) \}$.
2. $I = \{ (\hat{s}, f) \mid \hat{s} \in \hat{I} \}$.
3. $R((\widehat{s_0}, f_0), (\widehat{s_1}, f_1))$ iff $\hat{R}(\widehat{s_0}, \widehat{s_1})$.
4. $L((\hat{s}, f)) = f$.
5. Each $(\hat{P}, \hat{Q}) \in \hat{F}$ is transformed into a corresponding pair

$$(\{ (\hat{s}, f) \mid \hat{s} \in \hat{P} \}, \{ (\hat{s}, f) \mid \hat{s} \in \hat{Q} \})$$

   in F.

Example 4.6 Figure 4.3 shows the structure M corresponding to a traffic light. The structure has one state component c (for "color") which can take on one of the values $\{red, yellow, green\}$. The labels in the figure indicate the value of c in the different states. We abbreviate *red* by r, *yellow* by y, and *green* by g in the figure. The structure also has an acceptance condition requiring that we not loop forever in the state where $c = red$. The abstract state component corresponding



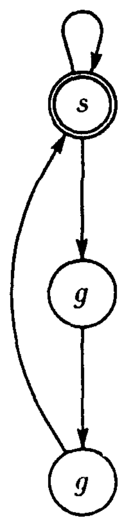
GF$(c \neq red)$

Figure 4.3: A structure representing a traffic light

to c will be denoted by $\hat{c}$. It will range over the values $\{stop, go\}$, and we will use the abstraction defined by $h(red) = stop$ and $h(yellow) = h(green) = go$. With this mapping, $abs_u(M)$ is shown in figure 4.4. In the figure, s indicates *stop* and g denotes *go*. The acceptance condition carries over as well: infinitely often, we must visit one of the bottom two states (where $\hat{c} = go$). On the other hand, if we let $\widehat{M}$ be the structure in figure 4.4, then we can also apply abs_l to $\widehat{M}$. This process is shown in figure 4.5. In the figure, the dashed arrows indicate the mapping between abstract-level states and concrete-level states. The lower two abstract states each map to a pair of concrete states. Note that the resulting structure $abs_l(abs_u(M))$ can simulate M, as implied by the definition of a conservative connection. □

Theorem 4.1 (abs_u, abs_l) is a conservative connection.

The proof of this is deferred; here, we just give the intuition. Suppose we know that $abs_u(M) \preceq \widehat{M}$. Given a state s of M, we lift it to the abstract level using abs_u. Now at this level, s can be simulated by



GF$(\hat{c} \neq stop)$

Figure 4.4: The result of applying abs_u to the structure in figure 4.3

some state $\hat{s}$ of $\widehat{M}$. However, each state $\hat{s}$ of $\widehat{M}$ can be viewed as a set of states at the concrete level, one for each possible concrete labeling function f satisfying $\widehat{L}(\hat{s}) = h(f)$. Thus, $abs_l(\widehat{M})$ will have a state $(\hat{s}, L(s))$, and this state will be able to simulate s.

It is easy to see that abs_u and abs_l are both monotonic with respect to $\preceq$. They can also be pushed over composition. For abs_u, every state of $abs_u(M \parallel M')$ is also a state (s, s') of $M \parallel M'$. This means that s and s' are states in $abs_u(M)$ and $abs_u(M')$, respectively, and they have compatible labelings. Hence (s, s') is also a state of $abs_u(M) \| abs_u(M')$, and this state can simulate (s, s') in $abs_u(M \parallel M')$. For abs_l, a state $(\hat{s}, \hat{s}')$ in $\widehat{M} \parallel \widehat{M'}$ gives rise to states $((\hat{s}, \hat{s}'), f)$ in $abs_l(\widehat{M \parallel M'})$. Now $(\hat{s}, f)$ and $(\hat{s}', f)$ must be states of $abs_l(\widehat{M})$ and $abs_l(\widehat{M'})$ respectively, and so $((\hat{s}, f), (\hat{s}', f))$ is a state of their composition. This state can be seen to simulate $(\hat{s}, \hat{s}')$.

Note $abs_u(M)$ is essentially like M, but with the labeling function changed. In order to reduce the complexity of verification, we will generally apply *collapse* immediately after abs_u. However, constructing

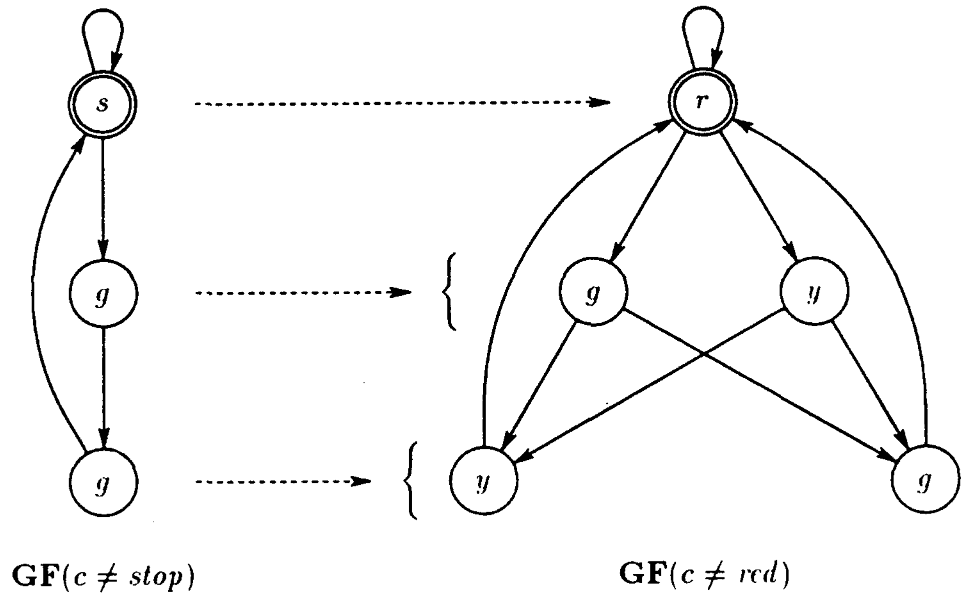


Figure 4.5: The result of applying abs_l to the structure in figure 4.4

M in order to compute $abs_u(M)$ and then $collapse(abs_u(M))$ is often not practical. We address this problem in the next section.

4.2 Computing Abstractions

We use two methods to avoid having to examine M. The first is to use the fact that M is often given as a composition. By pushing the approximation computation over the composition, we do not have to construct the product state space of the parts. The other technique relies on the fact that we usually have an implicit representation for M. For example, suppose M is given by a program in a finite-state language. By using a nonstandard semantics for the language, we can directly compile an approximation to $collapse(abs_u(M))$. This approach is similar to the use of abstract interpretation in program analysis [40, 41] and was first applied to verification by Clarke, Grumberg, and Long [30]. We now illustrate the details of this process using a simple finite state language which we call $\mathcal{L}_0$. Programs in $\mathcal{L}_0$ can be used to describe structures, but we emphasize that $\mathcal{L}_0$ is intended only for illustration purposes: it does not contain facilities that would be needed in a practical language. After discussing the syntax and intuitive meanings of $\mathcal{L}_0$ programs, we will give two semantics: a standard one, and one that can be used to produce an approximation to the abstracted structure.

Definition 4.5 The *textual classes for the language* $\mathcal{L}_0$ are defined as follows:

1. Variables: $v_0, v_1, \ldots$
2. Functions and constants: $f_0, f_1, \ldots$
3. Expressions: an expression e is either a variable reference v_i or a function invocation $f_i(e_0, \ldots, e_{n-1})$.
4. Statements: a statement s has one of the following forms:
   (a) an assignment statement $v_i := e$;
   (b) a conditional statement $e_0 \rightarrow s_0 \mid \ldots \mid e_{n-1} \rightarrow s_{n-1}$; or

(c) a sequential composition $s_0; \ldots ; s_{n-1}$; or

(d) a parallel composition $s_0 \parallel \ldots \parallel s_{n-1}$.

For conditionals, we require that the union of the guards be total (their disjunction must be a tautology), so one alternative is always selected. In the composition, we require that different s_i do not change the same variable, as this may lead to conflicts. To avoid this, we define a function *changes* that gives the set of variables changed by a statement. Then we must have $\mathit{changes}(s_i) \cap \mathit{changes}(s_j) = \emptyset$ for $i \neq j$. Formally, *changes* is defined as follows:

(a) $\mathit{changes}(v_i := e) = \{v_i\}$.

(b) $\mathit{changes}(e_0 \rightarrow s_0 \mid \ldots \mid e_{n-1} \rightarrow s_{n-1}) = \bigcup_{i=0}^{n-1} \mathit{changes}(s_i)$.

(c) $\mathit{changes}(s_0; \ldots ; s_{n-1}) = \bigcup_{i=0}^{n-1} \mathit{changes}(s_i)$.

(d) $\mathit{changes}(s_0 \parallel \ldots \parallel s_{n-1}) = \bigcup_{i=0}^{n-1} \mathit{changes}(s_i)$

Both of these restrictions can be eliminated, but since $\mathcal{L}_0$ is only being used for illustrative purposes, we choose to keep things simple.

5. Programs: a program is a pair of statements $s_{\text{init}}; s_{\text{trans}}^{\omega}$. The statement s_{init} is used to set up the initial states from which the program begins execution. At that point, we proceed by executing s_{trans} repeatedly. (Thus the notation: the ω is intended to suggest infinite execution of s_{trans} following one execution of s_{init}.) To derive the actual set of initial states, we execute s_{init} starting from an arbitrary state; any state that is reached as a result is an initial state.

The state space of an $\mathcal{L}_0$ program will be a set of tuples of valuations over a collection $A = \{a_0, a_1, a_2, \ldots\}$ of state components. The variable v_i within a program is used to refer to the value of component a_i within a state, or to specify how the value of that component changes. Note that we have not specified the operators that are allowed in expressions in an $\mathcal{L}_0$ program, but the exact ones are not important.

Before giving a formal semantics for $\mathcal{L}_0$, we start with an intuitive description. Expressions will have their usual meanings. An assignment statement $v_i := e$ sets the value of the component a_i to the result of evaluating e. To execute a conditional $e_0 \rightarrow s_0 \mid \ldots \mid e_{n-1} \rightarrow s_{n-1}$, we evaluate all of the expressions e_i, each of which should yield a boolean value. Next, we choose an i for which e_i is true (there must be at least one), and then execute the corresponding s_i. Multiple e_i being true gives rise to nondeterminism. For a sequential composition $s_0; \ldots ; s_{n-1}$, we execute the s_i in order. s_{i+1} is executed starting from the state where s_i finished. To execute the parallel composition $s_0 \parallel \ldots \parallel s_{n-1}$, we first execute each s_i starting from the current state. Then, we merge the result of each of these executions to obtain the result of executing the parallel composition. The merging is done as follows: if s_i sets the value of state component a_j to the value of e, then the value of a_j after execution of the parallel composition will be the value of e. In order to ensure that different s_i do not set the same a_j to conflicting values, we *require that different s_i cannot assign* to the same variable. This is the reason for introducing the function *changes* above.

Example 4.7 Consider the Collatz problem (the "$3x + 1$ problem"). You are given a natural number x and asked to apply the following procedure. If x is odd, multiply it by three and add one; if it is even, divide it by two. If this procedure is repeated continually, will you always reach $x = 1$? (The answer to this question is currently unknown.) An $\mathcal{L}_0$ program that executes steps of the $3x+1$ problem for the initial value 42 is shown in figure 4.6. We will come back to this program when we consider the process of direct abstract-level compilation. □

We now proceed to give the formal semantics of $\mathcal{L}_0$. Since we are interested in producing initial state and transition relations, a relational semantics is most natural. For simplicity, we will assume that the set of state components (and corresponding variables) is fixed, and that the domains of values for these components is likewise fixed. In a practical language of course, these would be specified within the program. We also ignore type checking issues: a given state component can only hold certain values, and assignments to the corresponding variable must respect this. In order to give semantics for conditionals, we need to be

```
INITIAL
x := 42

TRANSITIONS
  even?(x) -> x := x/2
| odd?(x)  -> x := x+x+x;
              x := x+1
```

Figure 4.6: Example $\mathcal{L}_0$ program

able to specify that an expression evaluates to *true*. For simplicity, we assume that *true* is a special data value, and that it is left fixed by the abstraction mapping.

The semantics will be in terms of a meaning function, denoted as $[\![\cdot]\!]$, which we take as assigning meanings to expressions, statements, and programs. The meaning of an expression will be a function that takes a state of the system and returns the value of that expression when evaluated at that state. Following standard notational conventions, we write this in curried form: $[\![e]\!]\sigma$ means take e, find its meaning (a function from states to values), and apply this function to the state σ. States of the system are viewed as valuations, i.e., mappings from variables to values. The meaning of a statement is a relation between states that is true iff executing the statement starting in the first state can result in the second state. If the statement s can take us from state σ to state σ', we write $[\![s]\!](\sigma, \sigma')$. The meaning of a program will be a structure. The semantics are parameterized by concrete functions that correspond to the operators appearing in the expressions.

Definition 4.6 The *standard semantics for* $\mathcal{L}_0$ (over concrete functions $f_0, f_1, \ldots$) is defined as follows:

1. Expressions:

   (a) The meaning of a variable in a particular state is just the value of the variable in the state:

$$[\![v_i]\!]\sigma = \sigma(v_i).$$

(b) The meaning of a function invocation $f_i(e_0, \ldots, e_{n-1})$ is the result of first evaluating the meaning of each e_i (in σ) and then applying f_i to the result:

$$[\![f_i(e_0, \ldots, e_{n-1})]\!]\sigma = f_i([\![e_0]\!]\sigma, \ldots, [\![e_{n-1}]\!]\sigma).$$

2. **Statements:**

(a) An assignment statement $v_i = e$ takes us between the states σ and σ' when σ' is obtained from σ by first evaluating the expression e in the state σ and then setting the value of v_i in σ' to the result: $[\![v_i := e]\!](\sigma, \sigma')$ iff $\sigma' = \sigma[[\![e]\!]\sigma / v_i]$.

(b) For a conditional, we evaluate all of the guards in the state σ, choose one which is *true*, and then execute the corresponding statement to take us between σ and σ':

$$[\![e_0 \rightarrow s_0 \mid \ldots \mid e_{n-1} \rightarrow s_{n-1}]\!](\sigma, \sigma')$$

iff there exists i such that

$$([\![e_i]\!]\sigma = \mathit{true}) \wedge [\![s_i]\!](\sigma, \sigma').$$

(c) For a sequential composition, we just execute each statement in turn.

$$[\![s_0; \ldots; s_{n-1}]\!](\sigma, \sigma')$$

iff there exists $\sigma_0, \ldots, \sigma_n$ such that $\sigma_0 = \sigma$, $\sigma_n = \sigma'$, and for all $0 \leq i < n$, $[\![s_i]\!](\sigma_i, \sigma_{i+1})$.

(d) In a parallel composition, recall that we have a syntactic restriction that two different statements in the composition cannot change the same variable. Thus, to get the effect of parallel execution, we just execute each statement in the composition starting from the state σ. Then we fold all of the changes that the statements make together to get σ'. Because of the above restriction, we cannot run into conflicts when doing the merging.

$$[\![s_0 \parallel \ldots \parallel s_{n-1}]\!](\sigma, \sigma')$$

iff there exists $\sigma_0, \ldots, \sigma_{n-1}$ such that $[\![s_i]\!](\sigma, \sigma_i)$ and:

i. $\sigma'(v_j) = \sigma_i(v_j)$ when there exists a (unique) i such that $v_j \in changes(s_i)$;

ii. $\sigma'(v_j) = \sigma(v_j)$ otherwise.

3. Programs: The meaning of a program $s_{\text{init}}; s^{\omega}_{\text{trans}}$ is the following structure.

   (a) S is the set of all valuations σ.

   (b) For the initial states, we execute s_{init} from an arbitrary state. Thus, $\sigma' \in I$ iff there exists σ such that $[\![s_{\text{init}}]\!](\sigma, \sigma')$.

   (c) The possible transitions are those that are allowed by s_{trans}: $R = [\![s_{\text{trans}}]\!]$.

   (d) The labeling of a state is just given by the state: $L(\sigma, a_i) = \sigma(v_i)$.

   (e) $F = \emptyset$.

We now turn to the problem of compiling an $\mathcal{L}_0$ program in order to obtain an approximation to the actual meaning of the program. We will assume that the value of the variable v_i is to be abstracted by the mapping h_i, i.e., h_i is a mapping from D_{a_i} (the domain for v_i) to $D_{\widehat{a}_i}$ (the abstract domain for this same variable). Now we want to work directly over abstract domain elements in order to avoid having to apply an abstraction such as abs_u after the compilation process. By working in the abstract domain, we generally lose information. As a result, we often cannot tell exactly what the value of an expression should be. For example, suppose the concrete domain that we are considering is the natural numbers, and say that the subtraction $m - n$ is defined to produce 0 when $m < n$. Also assume that the abstract value corresponding to a number is equal to the value of number modulo 5. Given just the values of m and n modulo 5, we cannot tell exactly what the value of $m - n$ modulo 5 will be. On the other hand, we do have some information: it must be either 0 (if $m < n$) or $m - n$ modulo 5 (if $m \geq n$). *We will capture this uncertainty by using a relation to represent the value of an expression.* When the relation corresponding to an expression is true for some abstract domain element, it intuitively indicates that the expression may evaluate to that abstract value. Of course, this uncertainty also appears at the level of the primitive operators that appear in expressions, and hence the semantics now will depend on a set of relations rather than on a set of functions as above. There will be a relation for each function, and we will denote the relation corresponding to f_i by P_{f_i}. While we want P_{f_i} to overestimate the possible values of f_i, we do not want to be too conservative. For example, while having P_{f_i} be the universal relation (i.e., saying that f_i could produce any value) would give a valid approximation, we would not be able to prove anything interesting by examining the abstract structure. Thus, we want to include only those values that are strictly necessary. This suggests the following: we take $P_{f_i}(\widehat{d_0}, \ldots, \widehat{d_{n-1}}, \hat{d})$ iff

$$\exists d_0 \ldots d_{n-1} d \, [\bigwedge_{i=0}^{n-1} h(d_i) = \hat{d}_i \wedge h(d) = \hat{d} \wedge f_i(d_0, \ldots, d_{n-1}) = d].$$

(Here, we are abusing notation and writing $h(d)$ for $d \in D_{a_i}$ to denote $h_i(d)$.) That is, P_{f_i} is true for $\widehat{d_0}, \ldots, \widehat{d_{n-1}}, \hat{d}$ when: given arguments whose abstract values are $\widehat{d_0}, \ldots, \widehat{d_{n-1}}$, f_i could produce a result whose abstract value is $\hat{d}$. Now we define the approximating semantics for $\mathcal{L}_0$ programs. Recall that we are now going to be compiling entirely at the abstract level.

Definition 4.7 The *upper approximating semantics for* $\mathcal{L}_0$ (over the relations $P_{f_0}, P_{f_1}, \ldots$) is denoted by $[\![\cdot]\!]_u$ and is defined as follows:

1. Expressions: Recall that the meaning of an expression will be a relation that is true for an abstract value $\hat{d}$ when it appears that the actual value d could be such that $h(d) = \hat{d}$.

   (a) The meaning of a variable reference v_i is a relation that is true for $\hat{d}$ when the actual value of v_i could map to $\hat{d}$. However, the abstract value of v_i is given by the state $\hat{\sigma}$. Thus, $([\![v_i]\!]_u\hat{\sigma})(\hat{d})$ iff $\hat{\sigma}(v_i) = \hat{d}$.

   (b) For a function application $f_i(e_0, \ldots, e_{n-1})$, we want to evaluate the arguments and then apply f_i. When we evaluate the argument e_i, we get the relation $[\![e_i]\!]_u\hat{\sigma}$, specifying the possible abstract values of e_i. Now P_{f_i} tells us the possible abstract values of f_i given a sequence of abstract inputs.

Thus, we simply look at all the possible sequences of abstract inputs and check P_{f_i} for each sequence.

$$(\llbracket f_i(e_0, \ldots, e_{n-1}) \rrbracket_u \widehat{\sigma})(\widehat{d})$$

iff

$$\exists \widehat{d_0} \ldots \widehat{d_{n-1}} \, [(\llbracket e_0 \rrbracket_u \widehat{\sigma})(\widehat{d_0}) \wedge \cdots \wedge (\llbracket e_{n-1} \rrbracket_u \widehat{\sigma})(\widehat{d_{n-1}}) \wedge P_{f_i}(\widehat{d_0}, \ldots, \widehat{d_{n-1}}, \widehat{d})].$$

2. Statements:

   (a) For an assignment $v_i := e$, we again just want to replace the next state value of v_i with the value of e. The possible values of e are given by the relation $\llbracket e \rrbracket_u \widehat{\sigma}$, so we just allow $\widehat{\sigma'}(v_i)$ to be any value satisfying this relation.

   $$\llbracket v_i := e \rrbracket_u (\widehat{\sigma}, \widehat{\sigma'})$$

   iff there exists $\widehat{d}$ such that $(\llbracket e \rrbracket_u \widehat{\sigma})(\widehat{d})$ ($\widehat{d}$ is a possible value of e) and $\sigma' = \sigma[\widehat{d}/v_i]$.

   (b) For a conditional, we want to evaluate each guard and then choose one which is *true*. However, we cannot necessarily tell the exact value of each guard. In order to simulate what the actual program might do, we allow execution of a statement s_i whenever the corresponding guard e_i *could* be true.

   $$\llbracket e_0 \rightarrow s_0 \mid \ldots \mid e_{n-1} \rightarrow s_{n-1} \rrbracket_u (\widehat{\sigma}, \widehat{\sigma'})$$

   iff there exists i such that $(\llbracket e_i \rrbracket_u \widehat{\sigma})(true)$ (e_i could be *true*) and $\llbracket s_i \rrbracket_u (\widehat{\sigma}, \widehat{\sigma'})$.

   (c) $\llbracket \cdot \rrbracket_u$ for sequential compositions and parallel compositions is defined in the same manner as $\llbracket \cdot \rrbracket$ back in definition 4.6. (This is because these operations do not directly involve evaluating expressions.)

3. Programs: The program $s_{\text{init}}; s^{\omega}_{\text{trans}}$ again evaluates to a structure $\widehat{M}$, but this time it is over abstract state components. Other than this, the definition is analogous to that for $\llbracket s_{\text{init}}; s^{\omega}_{\text{trans}} \rrbracket$.

(a) $\hat{S}$ is the set of all valuations $\hat{\sigma}$.

(b) $\hat{\sigma}' \in \hat{I}$ iff there exists $\hat{\sigma}$ such that $[s_{\text{init}}]_u(\hat{\sigma}, \hat{\sigma}')$.

(c) $\hat{R} = [s_{\text{trans}}]_u$.

(d) $\hat{L}(\hat{\sigma}, \hat{a}_i) = \hat{\sigma}(v_i)$.

(e) $\hat{F} = \emptyset$.

Now in order to be able to use our approximating semantics for verification purposes, we need that $abs_u([\![s_{\text{init}}; s^{\omega}_{\text{trans}}]\!]) \preceq [s_{\text{init}}; s^{\omega}_{\text{trans}}]_u$. We actually have the following stronger result, whose proof is deferred.

Theorem 4.2 Let $s_{\text{init}}; s^{\omega}_{\text{trans}}$ be an $\mathcal{L}_0$ program. Then

$$collapse(abs_u([\![s_{\text{init}}; s^{\omega}_{\text{trans}}]\!])) \preceq [s_{\text{init}}; s^{\omega}_{\text{trans}}]_u.$$

Since $M \preceq collapse(M)$ for all structures M, this implies

$$abs_u([\![s_{\text{init}}; s^{\omega}_{\text{trans}}]\!]) \preceq [s_{\text{init}}; s^{\omega}_{\text{trans}}]_u.$$

Example 4.8 Consider the program of example 4.7. Suppose that we abstract x by mapping even natural numbers to *even* and odd ones to *odd*. First, let us compute the P_{f_i} used in the program. We have predicates *odd?* and *even?* mapping natural numbers to booleans, and we have addition and integer division. Then, as expected, we get

$$P_{odd?} = \{(odd, true), (even, false)\}$$

and

$$P_{even?} = \{(odd, false), (even, true)\}.$$

Addition also behaves nicely:

$$\begin{aligned} P_+ = \{&(odd, odd, even), (even, even, even),\\ &\qquad (odd, even, odd), (even, odd, odd)\}.\end{aligned}$$

With division, however, we find that $P_/$ is the universal relation. (We also have the obvious relations representing the constants 1 and 2 that are used in the program.) Now we begin assigning meaning to the pieces of the program.

Consider the expression $x+1$. What is the meaning that the approximating semantics assigns to this expression? Recall that $[\![x+1]\!]_u\hat{\sigma}$ is supposed to be a relation representing possible abstract values of $x+1$ given that x has the abstract value $\hat{\sigma}(x)$. Let us consider the abstract value *odd* and determine when it can be in $[\![x+1]\!]_u\hat{\sigma}$. We have that *odd* is a possible value iff there exist $\widehat{d_0}$ and $\widehat{d_1}$ (chosen from $\{even, odd\}$) such that $P_+(\widehat{d_0}, \widehat{d_1}, odd)$ and $([\![x]\!]_u\hat{\sigma})(\widehat{d_0})$ and $P_1(\widehat{d_1})$. Since P_1 is only true for the abstract value *odd*, we must have $\widehat{d_1} = odd$. Then $P_+(\widehat{d_0}, odd, odd)$ is only true for $\widehat{d_0} = even$. Hence we must have $([\![x]\!]_u\hat{\sigma})(even)$, i.e., x must have the abstract value *even*, and so $\hat{\sigma}(x) = even$. In summary, we find that x must have the abstract value *even* for $x+1$ to evaluate to the abstract value *odd*. Similarly, x must be *odd* for $x+1$ to give *even*. Using the above, we can derive the relation $[\![x := x+1]\!]_u$. Recall that this relation tells us the possible abstract state changes that can occur when we execute $x := x+1$. If we identify a valuation by the value it assigns to x, then

$$[\![x := x+1]\!]_u = \{(odd, even), (even, odd)\}.$$

For $x := x+x+x$, we obtain

$$[\![x := x+x+x]\!]_u = \{(odd, odd), (even, even)\}.$$

Taking the relational product:

$$[\![x := x+x+x; x := x+1]\!]_u = \{(odd, even), (even, odd)\}.$$

For $x := x/2$, we get the universal relation. Evaluating the conditional, we obtain the final transition relation

$$\{(odd, even), (even, even), (even, odd)\}.$$

From this abstract compilation, we can tell that the system would satisfy the property: "if x is odd, then one step later, x will be even". □

In implementing the above ideas, the main difficulty is in producing the P_{f_i}. When performing a verification, the user must have a lot of flexibility in constructing abstractions. Contrast this with the situation where abstract interpretation is being used by a compiler to gather data-flow information for optimization purposes. Here, if the abstraction is not precise enough to prove that a particular optimization is safe, then the program will simply run a bit slower. In verification, when the user decides that the current abstraction is not precise enough to prove some property, she must have the flexibility to modify the abstraction in order to try to capture the information required. Obviously, making the user provide new P_{f_i} each time the abstraction changes is extremely tedious and error-prone. Also, we have found that we often need to make up new abstractions during the course of a verification. Hence, having a fixed "catalog" of allowed abstractions is not an option. The alternative is to have the user provide only the abstraction mapping (the h_j) for each variable and to let the compiler produce the P_{f_i} as needed. This requires the ability to evaluate the relational products

$$\exists d_0 \ldots d_{n-1} d \, [\bigwedge_{i=0}^{n-1} h(d_i) = \hat{d}_i \wedge h(d) = \hat{d} \wedge f_i(d_0, \ldots, d_{n-1}) = d]$$

automatically. In a BDD-based compiler, this is feasible: BDDs essentially give us a way for manipulating sets, relations, and functions over finite domains. This is the approach we used in developing the prototype compiler described in the next section.

4.3 Example Abstractions

In this section, we discuss some abstractions which have proved useful in practice. Each is illustrated with a small example. These examples are drawn from the paper by Clarke, Grumberg, and Long [30]. The examples will be given using a finite state language that is suitable for describing Moore machines. The main features of this language are:

1. It is procedural and contains a variety of structured programming constructs, such as `while` loops. Non-recursive procedures are also available.

2. It is finite state. The user must specify a fixed number of bits for each input and output in a program.

3. In keeping with the Moore machine semantics, the model of computation is a synchronous one. At the start of each time step, inputs to the program are obtained from the environment. All computation in a program is viewed as instantaneous (i.e., occurring in zero time). There is one special statement, `wait`, which is used to indicate the passage of time. When a `wait` statement is encountered, any changes to the program's outputs become visible to the environment, and a new time step is initiated. Thus, computation proceeds as follows: obtain inputs, compute (in zero time) until a `wait` is encountered, make output changes visible, obtain new inputs, etc. The `wait` statements indicate the control points in the program.

Aside from the `wait` statement, most of the language features used in the examples are self-explanatory. Additional features will be described in more detail as needed.

We implemented a prototype compiler to take programs written in the language and compile them down into Moore machines. During the compilation process, BDDs for the initial states and transitions of the program are produced by symbolic execution. When a program is compiled, the user may also specify abstractions for some of the inputs or outputs. These abstractions are given by simply specifying the functions h_i. By using the techniques described previously, the compiler directly generates an abstract Moore machine. There are a number of abstractions built into the compiler, some of which are described below. In addition, the user may define new abstractions by supplying procedures to build the BDDs representing the abstraction function. Abstract versions of the language primitives are computed automatically by the compiler as needed during the compilation. Since the language is much more complex than $\mathcal{L}_0$, we will not give its formal semantics or the approximating semantics here.

Figure 4.7 is a small example program, a settable countdown timer. The timer has two inputs, *set* and *start*, which are one and eight bits wide respectively. There are also two outputs: *count*, which is eight bits wide and is initially zero; and *alarm*, which is one bit and initially one. At each time step, the operation of the counter is as follows. If *set* is one, then the counter is set to the value of *start*. Otherwise, if the counter is not zero, it is decremented. The *alarm* output is set to one when *count* is zero, and to zero if *count* is nonzero.

```
input set[1];
input start[8];
output count[8] := 0;
output alarm[1] := 1

loop
  if set = 1
    count := start
  else if count > 0
    count := count-1
  endif;
  if count = 0
    alarm := 1
  else
    alarm := 0
  endif;
  wait
endloop
```

Figure 4.7: An example program

4.3.1 Congruence modulo an integer

For verifying programs involving arithmetic operations, a useful abstraction is congruence modulo a specified integer m:

$$h(i) = i \bmod m.$$

This abstraction is motivated by the following properties of arithmetic modulo m.

$$((i \bmod m) + (j \bmod m)) \bmod m \equiv i + j \pmod{m}$$
$$((i \bmod m) - (j \bmod m)) \bmod m \equiv i - j \pmod{m}$$
$$((i \bmod m)(j \bmod m)) \bmod m \equiv ij \pmod{m}$$

In other words, we can determine the value modulo m of an expression involving addition, subtraction and multiplication by working with the values modulo m of the subexpressions.

The abstraction may also be used to verify more complex relationships by applying the following result from elementary number theory.

Theorem 4.3 (Chinese remainder theorem) Let $m_1, m_2, \ldots, m_n$ be positive integers which are pairwise relatively prime. Define $m = m_1 m_2 \ldots m_n$, and let $b, i_1, i_2, \ldots, i_n$ be integers. Then there is a unique integer i such that

$$b \leq i \leq b+m \qquad \text{and} \qquad i \equiv i_j \pmod{m_j} \qquad \text{for } 1 \leq j \leq n.$$

Suppose that we are able to verify that at a certain point, the value of the nonnegative integer variable x is equal to i_j modulo m_j for each of the relatively prime integers m_1, m_2, ..., m_n. Further, suppose that the value of x is constrained to be less than $m_1 m_2 \ldots m_n$ (e.g., x is represented using k bits and $2^k < m_1 m_2 \ldots m_n$). Then using the above result, we can uniquely determine the value of x at that point from the i_j.

We illustrate this abstraction using a 16 bit by 16 bit unsigned multiplier (see figure 4.8). The program has inputs *req*, *in1* and *in2*. The last two inputs provide the factors to operate on, and the first is a request signal which starts the multiplication. Some number of time units later, the output *ack* will be set to true. At that point, either *output* gives the 16 bit result of the multiplication, or *overflow* is one if the multiplication overflowed. The multiplier then waits for *req* to become zero before starting another cycle. The multiplication itself is done with a series of shift-and-add steps. At each step, the low order bit of the first factor is examined; if it is one, then the second factor is added to the accumulating result. The first factor is then shifted right and the result is shifted left in preparation for the next step. One feature of the language which the program uses is the ability to extend an operand to a specified number of bits (lines 21 and 27), indicated using the colon operator. This facility is used to extend *output* and *factor2* when adding and shifting so that overflow can be detected. The statement

```
(overflow, output) := (output:17)+factor2
```

sets *output* to the 16 bit sum of *output* and *factor2* and *overflow* to the carry from this sum. Also, << is used to indicate left shift by the indicated number of bits, and right shifts are indicated with >>. The `break` statement is used to exit the innermost loop.

The specification we would like to use for the multiplier is a series of formulas of the following form.

$$\begin{aligned}&\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\mathit{in1} \bmod m = i) \wedge (\mathit{in2} \bmod m = j)\\&\qquad \rightarrow \mathbf{A}(\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack} \wedge (\mathit{overflow} \vee (\mathit{output} \bmod m = k))))\end{aligned}$$

Here, i and j range from 0 through $m-1$, $k = ij \bmod m$, and *waiting* is an atomic proposition which is true when execution is at line 13 in the program. Since verifying liveness properties such as those involving the until operator tends to be more complex than verifying safety properties, we will actually check the following weaker properties:

$$\begin{aligned}&\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\mathit{in1} \bmod m = i) \wedge (\mathit{in2} \bmod m = j)\\&\qquad \rightarrow \mathbf{A}(\neg \mathit{ack}\ \mathbf{W}\ \mathit{ack} \wedge (\mathit{overflow} \vee (\mathit{output} \bmod m = k)))).\end{aligned}$$

The operator $\mathbf{W}$ is the weak until operator; it is like $\mathbf{U}$, but the second argument is not required to ever become true. In general, $\mathbf{A}(f\,\mathbf{W}\,g)$ is equivalent to $\mathbf{A}(g\,\mathbf{V}\,f \vee g)$. We will later verify (using a different abstraction) that eventually an acknowledgment is always received. Then, using the tableau construction, the combination of these two properties can then be checked to imply the original specification.

To verify the properties described above, the input *in2* and the outputs *factor2* and *output* were all abstracted modulo m. The output *factor1* and its corresponding input *in1* were not abstracted, since the entire bit pattern of *factor1* is used to control when *factor2* is added to *output*. We performed the verification for $m = 5, 7, 9, 11$ and 32. These numbers are relatively prime, and their product, 110,880, is sufficient to cover all 2^{16} possible values of *output*. Now we would like to use theorem 4.3 to deduce the following class of properties:

$$\begin{aligned}&\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\mathit{in1} = i) \wedge (\mathit{in2} = j)\\&\qquad \rightarrow \mathbf{A}(\neg \mathit{ack}\ \mathbf{W}\ \mathit{ack} \wedge (\mathit{overflow} \vee (\mathit{output} = ij)))).\end{aligned}$$

In order to do this, we need to argue that we know the value of *output* modulo the different values of m at the same time point. Our property

```
input in1[16];
input in2[16];
input req;
output factor1[16] := 0;
output factor2[16] := 0;
output output[16] := 0;
output overflow := 0;
output ack := 0

procedure waitfor(e)
  while !e wait endwhile
endproc

loop
  waitfor(req);
  factor1 := in1; factor2 := in2;
  output := 0; overflow := 0; wait;
  loop
    if (factor1 = 0) | (overflow = 1)
      break
    endif;
    if factor1[0] = 1
      (overflow, output) := (output:17)+factor2
    endif;
    factor1 := factor1 >> 1; wait;
    if (factor1 = 0) | (overflow = 1)
      break
    endif;
    (overflow, factor2) := (factor2:17) << 1;
    wait
  endloop;
  ack := 1; wait;
  waitfor(!req);
  ack := 0
endloop
```

Figure 4.8: A 16 bit multiplier

was in fact chosen so that this is the case: we know something about the value of output at the point where *ack* is first asserted.

The entire verification required slightly less than 30 minutes of CPU time on a Sun 4. We also note that because the BDDs needed to represent multiplication grow exponentially with the size of the multiplier, it would not have been feasible to verify the multiplier directly. Further, even checking the above formulas on the unabstracted multiplier proved to be impractical. Note that the specification above admits the possibility that the multiplier always signals an overflow. We verified that this is not the case using the abstraction described in the next subsection.

4.3.2 Representation by logarithm

When only the order of magnitude of a quantity is important, it is sometimes useful to represent the quantity by (a fixed precision approximation of) its logarithm. For example, suppose $i \geq 0$. Define

$$\lg i = \lceil \log_2(i+1) \rceil,$$

i.e., $\lg i$ is 0 if i is 0, and for $i > 0$, $\lg i$ is the smallest number of bits needed to write i in binary. We take $h(i) = \lg i$.

As an illustration of this abstraction, consider again the multiplier of figure 4.8. Recall that a multiplier which always indicated an overflow would satisfy our previous specification. We note that if $\lg i + \lg j \leq 16$, then $\lg ij \leq 16$, and hence the multiplication of i and j should not overflow. Conversely, if $\lg i + \lg j \geq 18$, then $\lg ij \geq 17$, and the multiplication of i and j will overflow. When $\lg i + \lg j = 17$, we cannot say whether overflow should occur. These observations lead us to strengthen our specification to include the following two formulas.

$$\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\lg \mathit{in1} + \lg \mathit{in2} \leq 16) \rightarrow \mathbf{A}(\neg \mathit{ack}\, \mathbf{W}\, \mathit{ack} \wedge \neg \mathit{overflow}))$$
$$\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\lg \mathit{in1} + \lg \mathit{in2} \geq 18) \rightarrow \mathbf{A}(\neg \mathit{ack}\, \mathbf{W}\, \mathit{ack} \wedge \mathit{overflow}))$$

We represented all the 16 bit variables in the program by their logarithms. Compiling the program with this abstraction and checking the above properties required less than a minute of CPU time. We can also use this abstraction to verify that the program does eventually give an acknowledgment.

$$\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \rightarrow \mathbf{A}(\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack}))$$

Checking this required only a few seconds of CPU time. To ensure that we can in fact conclude the stronger specifications such as

$$\mathbf{AG}(\mathit{waiting} \wedge \mathit{req} \wedge (\lg \mathit{in1} + \lg \mathit{in2} \leq 16) \rightarrow \mathbf{A}(\neg \mathit{ack}\, \mathbf{U}\, \mathit{ack} \wedge \neg \mathit{overflow})),$$

we verified that

$$\mathbf{AG}(p_1 \wedge p_2 \rightarrow \mathbf{A}(\neg p_3\ \mathbf{W}\ p_3 \wedge p_4))$$

and

$$\mathbf{AG}(p_1 \rightarrow \mathbf{A}(\neg p_3\ \mathbf{U}\ p_3))$$

implies

$$\mathbf{AG}(p_1 \wedge p_2 \rightarrow \mathbf{A}(\neg p_3\ \mathbf{U}\ p_3 \wedge p_4)).$$

Instantiating p_1 with $\mathit{waiting} \wedge \mathit{req}$, p_3 with ack, and p_2 and p_4 as appropriate proves the desired properties.

4.3.3 Single bit and product abstractions

For programs involving bitwise logical operations, the following abstraction is often useful:

$$h(i) = \text{the } j\text{th bit of } i,$$

where j is some fixed number.

If h_1 and h_2 are abstraction mappings, then $h(i) = (h_1(i), h_2(i))$ also defines an abstraction mapping. Using this type of abstraction, it may be possible to verify properties that it is not possible to verify with either h_1 or h_2 alone.

As an example of using these types of abstractions, consider the program shown in figure 4.9. This program reads an initial 16 bit input and computes the parity of it. The output *done* is set to one when the computation is complete; at that point, *parity* has the result. The operator ^ used on line 8 denotes exclusive-or. Let $\sharp i$ be true if the parity of i is odd. One desired property of the program is the following.

1. The value assigned to b has the same parity as that of in; and
2. $\sharp b \oplus parity$ is invariant from that point onwards.

We can express the above with the following formula.

$$\neg\sharp in \wedge \mathbf{AX}(\neg\sharp b \wedge \mathbf{AG}\,\neg(\sharp b \oplus parity)) \vee \sharp in \wedge \mathbf{AX}(\sharp b \wedge \mathbf{AG}(\sharp b \oplus parity))$$

To verify this property, we used a combined abstraction for in and b. Namely, we grouped the possible values for these variables both by the value of their low order bit and by their parity. The verification required only a few seconds (note however, that this example is simple enough to check directly with a BDD-based verifier).

```
input in[16];
output parity[1] := 0;
output b[16] := 0;
output done[1] := 0

b := in;
wait;
while b != 0
  parity := parity ^ b[0];
  b := b >> 1;
  wait
endwhile;
done := 1
```

Figure 4.9: A parity computation program

In chapter 5, we will consider another very powerful type of abstraction. Now however, we turn to a method for abstracting the temporal behavior of a system.

4.4 Abstraction Via Observers

The abstractions defined previously give us a method for changing the set of values that a state component can take on. However, the abstract state component values were functions only of a single state in the unabstracted model. In this section, we consider a more general form of abstraction, which we call abstraction via observers. This type of abstraction makes it possible to have abstract state components that depend on the history of the computation.

Example 4.9 Consider a functional unit that receives some inputs, computes for some number of steps, and then gives an output. In a hardware implementation of such a device, pipelining is often used in order to increase throughput. A pipelined implementation might receive one set of inputs during each clock cycle and (after a suitable startup latency) give one output per cycle. That is, the behavior of the implementation over time is as follows:

Time 1	2	3	4	5	6
input	compute	compute	output		
	input	compute	compute	output	
		input	compute	compute	output

Suppose that we want to relate this to a specification that is given purely in terms of input/output behavior. That is, the timing of the specification is as follows:

Times 1 to 3	4	5	6
start up	input/output	input/output	input/output

Clearly, some method is needed for relating the timing of the implementation with that of the specification. This will be done via an *observer process*. An observer is a process that watches, but does not affect, some of the state components of the implementation. It has as outputs some of the state components of the specification. The composition of the observer with the implementation gives a specification-level view of the actions of the implementation. We will then compare this combined implementation/observer with the specification. Conversely, the specification may be combined with the observer to give an implementation-level view of the actions allowed by the specification. The observer process may have internal state that it uses to track what it has seen.

Let us make our example a bit more precise. Suppose that the functional unit reads a 16 bit input x and outputs a 16 bit result y. We will construct an observer process that watches x and y and produces as outputs $\widehat{x}$ and $\widehat{y}$ that correspond to the abstract-level I/O behavior. The observer must synchronize an input on x with the corresponding y output, and so it will store successive x inputs internally and only output them after a suitable delay. In contrast, it will pass y values to the abstract level immediately. The effect will be that at the abstract level, an $\widehat{x}$ value and its corresponding $\widehat{y}$ will appear simultaneously at the outputs of the observer. The actual observer process for this example is given by the program of figure 4.10. In the figure, the line

```
mealyoutput y_hat[16] := y;
```

is used to introduce a Mealy-type output (one that may depend on both inputs and internal state). In this case, the Mealy output **y_hat** is defined to be invariantly equal to the expression y, i.e., $\widehat{y}$ is always equal to the input y. □

```
input x[16];
internal x_internal_1[16];
internal x_internal_2[16];
output x_hat[16];
input y[16];
mealyoutput y_hat[16] := y;

loop
  x_hat := x_internal_2;
  x_internal_2 := x_internal_1;
  x_internal_1 := x;
  wait
endloop
```

Figure 4.10: Observer process for example 4.9

In the example, we mentioned that an observer should not affect the concrete level state components. To see why this is the case, suppose that in the example above, the observer blocks any attempt to give the implementation the input $x = 12$. Because of this, it will never output the value $\hat{x} = 12$. Now assume that our implementation works correctly for all values except $x = 12$, but for $x = 12$, it produces $y = 33$ instead of the correct $y = 44$. Then when we run our observer in parallel with the implementation, all of the pairs $(\hat{x}, \hat{y})$ that are observed at the abstract level are in fact correct. Hence, the (correct) abstract specification will be able to simulate this behavior, and we might erroneously conclude that the implementation is right. A similar problem can arise if the observer refuses to accept certain y values; in this case, the observer may suppress what would be an incorrect output by the implementation. We conclude that the observer must always be able to accept anything that might occur at the implementation level. (In the terminology of Dill, an observer must be *receptive*; the notion that we will use here corresponds to receptiveness in prefix-closed trace structures [43].) We now give the formal definition of an observer.

Definition 4.8 An *observer over a set of state components* A' is a structure M with the following properties:

1. The observer must be able to accept any initial value for the state components in A'. Formally, for every labeling function f over A', there exists $s \in I$ such that $f = L(s) \downarrow A'$.

2. The observer must be able to accept any change in the state components in A': for every labeling function f over A' and every state $s_0 \in S$, there exists s_1 such that $f = L(s_1) \downarrow A'$ and $R(s_0, s_1)$.

3. In order to avoid having the acceptance condition rule out some infinite sequences of concrete-level behaviors, we also require $F = \emptyset$. (The structure being abstracted may have acceptance conditions, but the observer may not; this is again a receptiveness issue.)

Now suppose that we are given a set of observers. To get the abstract-level view of M, we would like to just run the observers in parallel with M. However, there is still one other way that incorrect behavior by M can be suppressed. Suppose that we have two observers that both output the same abstract-level state component $\hat{x}$. If one wants to set $\hat{x} = 12$ and the other wants to set $\hat{x} = 13$, then the net effect is that they deadlock, and whatever implementation-level behavior lead up to this situation is effectively disallowed. This is again unacceptable, but we can avoid the problem by simply requiring that different observers do not both try to output the same abstract-level component. Note, however, that it is legal for multiple observers to watch the same component. With this restriction, we can now abstract the implementation by just composing with our observers and hiding the concrete state components.

Definition 4.9 Let $A = \{a_0, \ldots, a_{n-1}\}$, $\hat{A} = \{\widehat{a_0}, \ldots, \widehat{a_{n-1}}\}$, and suppose that $O = \{M_0, \ldots, M_{m-1}\}$ is a set of observers over A with $A_i \subseteq A \cup \hat{A}$. Also assume that for every pair M_i, M_j of observers with $i \neq j$, $A_i \cap A_j \cap \hat{A} = \emptyset$ (no two observers output the same abstract state component). Let $M' = M_0 \parallel \ldots \parallel M_{m-1}$. If M is a structure over A, then we define $obs_u(M)$ *(with respect to O)* to be $(M \parallel M') \downarrow \hat{A}$.

The map obs_u takes us from the concrete level to the abstract level. We want to produce a conservative connection, and at the moment we have the situation shown in figure 4.11. Assume that we are given $\widehat{M}$; what implementation-level behavior should this represent?

$$\begin{tikzcd}
(M \parallel M') \downarrow \hat{A} \arrow[r, dash] & \preceq \arrow[d, dotted, "\text{implies}"] \arrow[r, dash] & \widehat{M} \arrow[d, "?"] \\
M \arrow[u, "obs_u"] \arrow[r, dash] & \preceq \arrow[r, dash] & ?
\end{tikzcd}$$

Figure 4.11: Situation after defining obs_u

To answer this question, let us think about the composition M' of all of the observers in O. We can view this composition as telling us all of the abstract behaviors that would be observable if the implementation was completely nondeterministic and could do anything at any step. Now $\widehat{M}$ will generally not be consistent with all of these abstract behaviors. We can prune away the incompatible ones by simply composing M' with $\widehat{M}$. The result of this composition involves both concrete- and abstract-level state components, so we then eliminate the abstract components by restricting to A. This process of composition and restriction is the desired map obs_l.

Definition 4.10 Let A, $\hat{A}$, etc., be as in definition 4.9. If $\widehat{M}$ is a structure over $\hat{A}$, then we define $obs_l(\widehat{M})$ *(with respect to O)* to be $(\widehat{M} \parallel M') \downarrow A$.

We then have the following result, whose proof is deferred.

Theorem 4.4 (obs_u, obs_l) is a conservative connection.

Example 4.10 The type of abstraction given by abs_u and abs_l can be expressed using observers. There would be one observer for each abstraction function h_i. The observer for h_i simply looks at the value of state component a_i and immediately sets $\hat{a}_i$ to h_i of that value. In other words, we would be using a set of observers of the form shown in figure 4.12. Then (abs_u, abs_l) corresponds directly to (obs_u, obs_l). □

```
input a_i[16];
mealyoutput a_i_hat[16] := <h_i(a_i)>;

loop
  wait
endloop
```

Figure 4.12: Observer process corresponding to (abs_u, abs_l)

4.5 Summary

We have shown how abstraction can be used to simplify the process of checking $\preceq$. The basis for using abstraction is the notion of a conservative connection. The mappings in a conservative connection relate abstract-level and concrete-level structures, and if $\preceq$ holds at the abstract level, we can infer a similar relationship at the concrete level. We considered two main conservative connections. One was used for just for data abstraction. A more general one, abstraction via observers, allows us to abstract temporal behavior as well. In the case of data abstraction, we discussed a method for directly compiling abstract-level structures from a finite-state program and a user-supplied abstraction mapping. We implemented a compiler based on these ideas and used it to verify a number of examples.

4.6 Technical Details

We begin by sketching the proof that *collapse* is monotonic and can be distributed over compositions.

Proof We first prove monotonicity. Suppose $M \preceq M'$, with $A = A'$, and assume without loss of generality that every state of M and M' is reachable. Let $\widehat{M} = \mathit{collapse}(M)$ and $\widehat{M'} = \mathit{collapse}(M')$. Obviously, for every state s of M, there is a state s' of M' with $s \preceq s'$. Then $L(s) = L'(s')$, and so $\hat{S} \subseteq \widehat{S'}$. We claim that $\sqsubseteq$ defined by $\widehat{M}, \hat{f} \sqsubseteq \widehat{M'}, \hat{f}$ for all $\hat{f} \in \hat{S}$ is a simulation relation. Obviously related states agree on their labelings. Consider a path $\widehat{f_0}\widehat{f_1}\ldots$ in $\widehat{M}$. We must show that this same sequence is a path in $\widehat{M'}$. Since $\hat{R}(\hat{f_i}, \widehat{f_{i+1}})$, there exist s_i and s_{i+1} in M with $\hat{f_i} = L(s_i)$, $\widehat{f_{i+1}} = L(s_{i+1})$, and $R(s_i, s_{i+1})$. Now choose a state s_i' of M' with $s_i \preceq s_i'$. There must be a state s_{i+1}' of M' with $s_{i+1} \preceq s_{i+1}'$ and $R'(s_i', s_{i+1}')$. This implies $\widehat{R'}(\hat{f_i}, \widehat{f_{i+1}})$. The result must also satisfy the acceptance conditions, and so $\sqsubseteq$ is indeed a simulation relation. If $s \in I$, there must be $s' \in I'$ with $s \preceq s'$. This implies that $\sqsubseteq$ relates initial states in $\widehat{M}$ to initial states in $\widehat{M'}$. Hence we conclude $\widehat{M} \preceq \widehat{M'}$, i.e., that *collapse* is monotonic with respect to $\preceq$.

Let $M'' = M \parallel M'$, $\widehat{M} = collapse(M)$, $\widehat{M'} = collapse(M')$, and $\widehat{M''} = collapse(M'')$. Define the relation $\sqsubseteq$ to be

$$\{\, (L''((s,s')), (L(s), L'(s'))) \mid (s,s') \in S'' \,\}.$$

First, note that if $(s,s') \in S''$, then s and s' agree on the labeling for the state components in $A \cap A'$. Hence, $L(s)$ and $L'(s)$ agree on these same state components, and so $(L(s), L'(s'))$ is a state of $\widehat{M} \parallel \widehat{M'}$. $\sqsubseteq$ is in fact a simulation relation. Clearly, the states related by $\sqsubseteq$ have identical labeling functions. Let $\widehat{\pi''} = \widehat{f_0''}\widehat{f_1''}\widehat{f_2''}\ldots$ be a path in $\widehat{M''}$. By definition of *collapse*, for all i, there are states (s_i, s_i'), and (t_i, t_i') of M'' such that $R''((s_i, s_i'), (t_i, t_i'))$, $L''((s_i, s_i')) = \widehat{f_i''}$, and $L''((t_i, t_i')) = \widehat{f_{i+1}''}$. This implies $R(s_i, t_i)$ and $R'(s_i', t_i')$, and so $\widehat{R}(L(s_i), L(t_i))$ and $\widehat{R'}(L'(s_i'), L'(t_i'))$. Further, $L(s_i)$ and $L'(s_i')$ must agree on the labeling of state components in $A \cap A'$, as must $L(t_i)$ and $L'(t_i')$. Thus, $(L(s_i), L'(s_i'))$ and $(L(t_i), L'(t_i'))$ are states of $\widehat{M} \parallel \widehat{M'}$, and there is a transition between these states. This leads to a path in $\widehat{M} \parallel \widehat{M'}$ whose states are related by $\sqsubseteq$ to the states on $\widehat{\pi''}$. Thus, $\sqsubseteq$ is a simulation relation. Also, each initial state of $\widehat{M''}$ must have the form $L''((s,s'))$, where (s,s') is an initial state of M''. Now we find $(L(s), L'(s'))$ is a state of $\widehat{M} \parallel \widehat{M'}$, and $L(s)$ and $L'(s')$ are initial states since s and s' must be. Thus, $\sqsubseteq$ relates initial states to initial states, and so $\widehat{M''} \preceq \widehat{M} \parallel \widehat{M'}$; *collapse* does indeed distribute in the expected way over composition. □

We now turn to theorem 4.1, which states that (abs_u, abs_l) is a conservative connection.

Proof Let M and $\widehat{M'}$ be structures over A and $\widehat{A}$ with $abs_u(M) \preceq \widehat{M'}$, and define $\widehat{M} = abs_u(M)$, $M' = abs_l(\widehat{M'})$. Define $\sqsubseteq$ by $s \sqsubseteq (\widehat{s'}, f')$ iff $L(s) = f'$ and $s \preceq \widehat{s'}$. We prove that $\sqsubseteq$ is a simulation relation.

Obviously, states that are related by $\sqsubseteq$ have the same labeling. Suppose that $\pi = s_0 s_1 s_2 \ldots$ is a path in M from $s = s_0$ and that $s \sqsubseteq (\widehat{s'}, f')$. Notice that π is also a path in $\widehat{M}$. Since $s \preceq \widehat{s'}$, there must be a corresponding path $\widehat{\pi'} = \widehat{s_0'}\widehat{s_1'}\widehat{s_2'}\ldots$ from $\widehat{s'} = \widehat{s_0'}$ in $\widehat{M'}$. Thus, we have states $\widehat{s_i'}$ in $\widehat{M'}$ such that $s_i \preceq \widehat{s_i'}$ for all i. This implies $h(L(s_i)) = \widehat{L'}(\widehat{s_i'})$. Now let $f_i' = L(s_i)$; from the definition of M', we have that $(\widehat{s'_i}, f')$ is a state in M'. By the definition of $\sqsubseteq$, $s_i \sqsubseteq (\widehat{s'_i}, f')$. Hence we have a sequence

$$\pi' = (\widehat{s'_0}, L(s_0))(\widehat{s'_1}, L(s_1))(\widehat{s'_2}, L(s_2))\ldots$$

in M'. From the definition of the acceptance condition of M', it is easy to see that this is in fact a path, and since it corresponds to π, we conclude that $\sqsubseteq$ is a simulation relation.

If s is an initial state of M, then s is also an initial state of $\widehat{M}$. Since $\widehat{M} \preceq \widehat{M'}$, there is a corresponding initial state $\widehat{s'}$ of $\widehat{M'}$. As above, we find that $(\widehat{s'}, L(s))$ is an initial state of M'. Hence $\sqsubseteq$ relates initial states to initial states, and so $M \preceq M'$. □

The proof that the approximating semantics ($[\![\cdot]\!]_u$) produces a valid abstract-level model (theorem 4.2) is essentially a large induction on the structure of expressions and statements.

Proof Let $M = [\![p]\!]$, $\widehat{M} = collapse(abs_u(M))$, and $\widehat{M'} = [\![p]\!]_u$. We first note that $\widehat{S}$ and $\widehat{S'}$ are isomorphic. The former are labeling functions over $\widehat{A}$. The latter are valuations mapping variables v_i to elements in $D_{\widehat{a}_i}$, and each variable v_i has its associated $\widehat{a}_i$. Further, the state labeling functions for isomorphic states are the same. In $\widehat{S}$, it is simply the state itself. In $\widehat{S'}$, it maps $\widehat{a}_i$ to the value of v_i under the valuation that is the state. Let ϕ be this isomorphism: $\phi(\widehat{f})$ will be the valuation mapping v_i to $\widehat{f}(\widehat{a}_i)$. We extend ϕ to sets and relations in the natural way. If we can demonstrate that $\phi(\widehat{R}) \subseteq \widehat{R'}$ and $\phi(\widehat{I}) \subseteq \widehat{I'}$, then this will obviously be sufficient to prove $\preceq$. Now $\widehat{I}$ is the image of the set of labeling functions for initial states of M under h. Applying ϕ transforms these labeling functions back to valuations mapping each v_i to something in $D_{\widehat{a}_i}$. Now the states of M are valuations mapping variables to domains D_{a_i}. Let σ be such a valuation; we write $h(\sigma)$ to denote the valuation mapping v_i to $h_i(\sigma(v_i))$. Using this notation, we see $\phi(\widehat{I})$ is just $h(I)$. Similarly, $\phi(\widehat{R}) = h(R)$. Thus, we want to prove $h(R) \subseteq \widehat{R'}$ and $h(I) \subseteq \widehat{I'}$.

To do this, it is enough to show that for every $\mathcal{L}_0$ statement s, $h([\![s]\!]) \subseteq [\![s]\!]_u$. The proof here will proceed by induction on the structure of s. For some of the cases, we will need an auxiliary result relating the value of an expression under the two semantics. Let σ be a valuation mapping v_i to an element of D_{a_i}. Claim: for every $\mathcal{L}_0$ expression e, $h([\![e]\!]\sigma) \in [\![e]\!]_u(h(\sigma))$. To see this, we proceed by induction on the structure of e.

1. Suppose e is a variable v_i. Then $[\![e]\!]\sigma = \sigma(v_i)$, and $h([\![e]\!]\sigma) = h_i(\sigma(v_i))$. $[\![e]\!]_u(h(\sigma))$ is true for $\hat{d} \in D_{\hat{a}_i}$ iff $(h(\sigma))(v_i) = \hat{d}$. Now $(h(\sigma))(v_i) = h_i(\sigma(v_i)) = h([\![e]\!]\sigma)$, and thus the result holds in this case.

2. Assume e is $f_i(e_0, \ldots, e_{n-1})$. $h([\![e]\!]\sigma) = h(f_i([\![e_0]\!]\sigma, \ldots, [\![e_{n-1}]\!]\sigma))$. On the other hand, $[\![e]\!]_u(h(\sigma))$ is true for $\hat{d}$ iff

$$\exists \widehat{d_0} \ldots \widehat{d_{n-1}}\, [([\![e_0]\!]_u h(\sigma))(\widehat{d_0}) \wedge \cdots \wedge ([\![e_{n-1}]\!]_u h(\sigma))(\widehat{d_{n-1}}) \\ \wedge P_{f_i}(\widehat{d_0}, \ldots, \widehat{d_{n-1}}, \hat{d})].$$

Let $d_i = [\![e_i]\!]\sigma$, and take $\hat{d}_i = h(d_i)$. By the induction hypothesis, $h([\![e_i]\!]\sigma) \in [\![e_i]\!]_u(h(\sigma))$ for all i. Hence $h(d_i) \in [\![e_i]\!]_u(h(\sigma))$, $\hat{d}_i \in [\![e_i]\!]_u(h(\sigma))$. Recall that P_{f_i} is given by $P_{f_i}(\widehat{d_0}, \ldots, \widehat{d_{n-1}}, \hat{d})$ iff

$$\exists d_0 \ldots d_{n-1} d\, [\bigwedge_{i=0}^{n-1} h(d_i) = \hat{d}_i \wedge h(d) = \hat{d} \wedge f_i(d_0, \ldots, d_{n-1}) = d].$$

Let $d = f_i([\![e_0]\!]\sigma, \ldots, [\![e_{n-1}]\!]\sigma)$, i.e., $h([\![e]\!]\sigma) = h(d)$, and set $\hat{d} = h(d)$.

At this point, we have $d_0, \ldots, d_{n-1}, d$ with corresponding abstract values $\widehat{d_0}, \ldots, \widehat{d_{n-1}}, \hat{d}$. We know $h([\![e]\!]\sigma) = \hat{d}$. We also know $\hat{d}_i \in [\![e_i]\!]_u(h(\sigma))$. From the definition of P_{f_i}, and the fact that $d = f(d_0, \ldots, d_{n-1})$, we see that $P_{f_i}(\widehat{d_0}, \ldots, \widehat{d_{n-1}}, \hat{d})$. Hence $\hat{d} \in [\![e]\!]_u(h(\sigma))$, which is the desired result.

We now proceed to the induction on statements. Recall that we are trying to prove $h([\![s]\!]) \subseteq [\![s]\!]_u$ for all statements s.

1. Consider an assignment $v_i := e$. $[\![s]\!](\sigma, \sigma')$ iff $\sigma' = \sigma[[\![e]\!]\sigma/v_i]$. $[\![s]\!]_u(\hat{\sigma}, \hat{\sigma}')$ iff $\hat{d} \in [\![e]\!]_u\hat{\sigma}$ and $\hat{\sigma}' = \hat{\sigma}[\hat{d}/v_i]$. To show $h([\![s]\!]) \subseteq [\![s]\!]_u$, we assume $[\![s]\!](\sigma, \sigma')$ and prove $[\![s]\!]_u(h(\sigma), h(\sigma'))$. Let $\hat{\sigma} = h(\sigma)$ and $\hat{\sigma}' = h(\sigma')$. Obviously $\hat{\sigma}$ and $\hat{\sigma}'$ can differ only on the value

for v_i. Set $d = [\![e]\!]\sigma$ and take $\hat{d} = h(d)$. Then $(h(\sigma'))(v_i) = h(d) = \hat{d}$. We must show $\hat{d} \in [\![e]\!]_u\hat{\sigma}$. However, $[\![e]\!]_u\hat{\sigma} = [\![e]\!]_u(h(\sigma))$. By the above subresult, $\hat{d} = h(d) = h([\![e]\!]\sigma) \in [\![e]\!]_u(h(\sigma))$.

2. Suppose s is the conditional $e_0 \rightarrow s_0 \mid \ldots \mid e_{n-1} \rightarrow s_{n-1}$. We have $[\![e_0 \rightarrow s_0 \mid \ldots \mid e_{n-1} \rightarrow s_{n-1}]\!](\sigma, \sigma')$ iff there exists i such that $[\![e_i]\!]\sigma = true$ and $[\![s_i]\!](\sigma, \sigma')$. Also, $[\![e_0 \rightarrow s_0 \mid \ldots \mid e_{n-1} \rightarrow s_{n-1}]\!]_u(\hat{\sigma}, \hat{\sigma'})$ iff there exists i such that $([\![e_i]\!]_u\hat{\sigma})(true)$ and $[\![s_i]\!]_u(\hat{\sigma}, \hat{\sigma'})$. Again, we assume $[\![s]\!](\sigma, \sigma')$ and prove $[\![s]\!]_u(h(\sigma), h(\sigma'))$. Define $\hat{\sigma} = h(\sigma)$ and $\hat{\sigma'} = h(\sigma')$. By the induction hypothesis, if $[\![s_i]\!](\sigma, \sigma')$, then $[\![s_i]\!]_u(\hat{\sigma}, \hat{\sigma'})$. Thus, we just need to know that if $[\![e_i]\!]\sigma = true$, then $([\![e_i]\!]_u\sigma)(true)$. By the previous subresult, if e_i evaluates to *true*, then $h(true) \in [\![e_i]\!]_u(h(\sigma))$. But we also assumed that *true* was not abstracted ($h(true) = true$). Hence $([\![e_i]\!]_u(h(\sigma)))(true)$, as required.

3. For a sequential or parallel composition, the result follows in a straightforward manner from the induction hypothesis. □

Finally, we prove theorem 4.4: that (obs_u, obs_l) is a conservative connection.

Proof Let M be a structure over A, $\widehat{M} = obs_u(M)$, $\widehat{M''}$ be a structure over $\hat{A}$, and $M'' = obs_l(\widehat{M''})$, and suppose $\widehat{M} \preceq \widehat{M''}$. Define $\sqsubseteq$ by $s \sqsubseteq (\widehat{s''}, s')$ iff $L(s) = L''((\widehat{s''}, s'))$ and $(s, s') \preceq \widehat{s''}$. We show that $\sqsubseteq$ is a simulation relation between M and M''.

Obviously states related by $\sqsubseteq$ have identical labelings. Suppose $s \sqsubseteq (\widehat{s''}, s')$. Let $\pi = s_0 s_1 s_2 \ldots$ be a path from $s = s_0$ in M. By the definition of observer, there must be a path $(s_0, s'_0)(s_1, s'_1)\ldots$ in $\widehat{M}$, where $s' = s'_0$. We must have $(s, s') \preceq \widehat{s''}$, and so there is a path $\widehat{\pi''}$ of the form $\widehat{s''_0}\widehat{s''_1}\widehat{s''_2}\ldots$ from $\widehat{s''} = \widehat{s''_0}$ in $\widehat{M''}$ with $(s_i, s'_i) \preceq \widehat{s''_i}$ for all i. This implies that s'_i and $\widehat{s''_i}$ have identical labelings on $\hat{A}$, and so $(\widehat{s''_i}, s'_i)$ is a state of M'' for all i. Further, $L(s_i) = L''((\widehat{s''_i}, s'_i))$ since (s_i, s'_i) is a state of $\widehat{M}$. Hence $s_i \sqsubseteq (\widehat{s''_i}, s'_i)$ for all i, and so $\sqsubseteq$ is indeed a simulation relation.

Now let s be an initial state of M. By the definition of observer, there is some $s' \in I'$ such that (s, s') is an initial state of $\widehat{M}$. Since

$\widehat{M} \preceq \widehat{M''}$, there must be a corresponding state $\widehat{s''}$ of $\widehat{M''}$. Now $(\widehat{s''}, s')$ is an initial state of M'', and $L''((\widehat{s''}, s')) = L(s)$. This implies $s \sqsubseteq (\widehat{s''}, s')$, i.e., $\sqsubseteq$ relates initial states to initial states. Hence $M \preceq M''$. $\square$

We also note that both obs_u and obs_l are monotonic and can be pushed over composition. Consider, for example, obs_u. Monotonicity is straightforward: if M' is the composition of the observers, then $M_1 \preceq M_2$ implies $M_1 \parallel M' \preceq M_2 \parallel M'$, and restricting both sides to $\hat{A}$ also preserves $\preceq$. When we push obs_u over composition, we want to compare $((M_1 \parallel M_2) \parallel M') \downarrow \hat{A}$ with $((M_1 \parallel M') \downarrow \hat{A}) \parallel ((M_2 \parallel M') \downarrow \hat{A})$. To prove that the latter can simulate the former, we show that

$$\{\, (((s_1, s_2), s'), ((s_1, s'), (s_2, s'))) \mid s_1 \in S_1, s_2 \in S_2, s' \in S' \,\}$$

is a simulation relation.

Chapter 5

Symbolic Parameters

The dramatic effect of using BDDs to implement traditional verification algorithms is well-documented [4, 23, 37, 48, 67, 89]. However, they can also be used to add powerful new extensions to these methods. This additional power arises because BDDs give us a flexible and efficient facility for manipulating sets and relations over finite domains. In this chapter, we indicate some of the ways that this facility can be used.

5.1 First-Order Quantification

We extend CTL (definition 2.1) to include first-order quantification operators. To do this, we first allow the atomic formulas to mention variables that range over data values. We will assume that each variable is associated with some particular domain of values.

Definition 5.1 The logic *QCTL ("Quantified CTL")* over a set of state components A is the set of formulas given by the following inductive definition:

1. The constant *true* is a formula.

2. For each state component a in A, element d of D_a, and variable x ranging over values in D_a, $a = d$, $a = x$, and $x = d$ are formulas.

3. If φ and ψ are formulas, then $\neg\varphi$ and $\varphi \wedge \psi$ are formulas.

4. If φ and ψ are formulas, then $\mathbf{AX}\,\varphi$, $\mathbf{A}(\varphi\,\mathbf{V}\,\psi)$ and $\mathbf{A}(\varphi\,\mathbf{U}\,\psi)$ are formulas.

5. If φ is a formula, then so is $\forall x\,\varphi$.

We use the usual abbreviations; also $\exists x\,\varphi$ denotes $\neg\forall x\,\neg\varphi$.

The semantics of these formulas over structures is essentially the same as standard CTL, except parameterized by a valuation for the individual variables.

Definition 5.2 Let M be a structure and φ be a formula with $A \supseteq comp(\varphi)$. *Satisfaction of φ by a state s of M with respect to a valuation σ* for the individual variables in φ ($M, s, \sigma \models \varphi$) is defined as follows:

1. Satisfaction for *true*, $\neg\varphi$, $\varphi\wedge\psi$, $\mathbf{AX}\,\varphi$, etc., are defined essentially as in satisfaction of CTL formulas (definition 2.4).

2. $M, s, \sigma \models a = d$ iff $L(s, a) = d$. $M, s, \sigma \models a = x$ iff $L(s, a) = \sigma(x)$. $M, s, \sigma \models x = d$ iff $\sigma(x) = d$.

3. $M, s, \sigma \models \forall x\,\varphi$ iff for every d in the domain D_a associated with x, $M, s, \sigma[d/x] \models \varphi$.

M satisfies the formula φ if for every initial state s of M and valuation σ for the individual variables, $M, s, \sigma \models \varphi$. (Thus, free variables in φ are treated as being under the scope of a universal quantifier.)

At first glance, model checking for QCTL would seem to be an inefficient prospect. Whenever we encounter a subformula $\forall x\,\varphi$, we may have to check φ for each possible value of x. Naturally, the situation is worse when quantifiers are nested. Overall, since the model checking problem for QCTL obviously subsumes the satisfiability problem for quantified boolean formulas (QBF) [58], we cannot expect an algorithm that is polynomial time in the size of our formula. Consider the situation in practice however. A natural use for QCTL is to describe systems that handle data. For example, if we are verifying a protocol and wish to specify that whatever data is sent is eventually received, we might use the following formula:

$$\mathbf{AG}\,\forall x\,(send \wedge senddata = x \rightarrow \mathbf{AF}(rcv \wedge rcvdata = x)).$$

The implementation probably behaves in a similar fashion regardless of what the data is. As a result, once we have verified that the formula holds for one particular data value, we expect that it will hold for all the others as well. If we could argue that the value that we picked is somehow representative of an arbitrary value, we might be able to avoid having to check them explicitly. Unfortunately, making this precise is difficult, especially if the implementation does have some data dependent behavior. Suppose, for example, that the implementation computes the parity of the data that is sent. In this case, it may not be enough to check just one data value, but we probably could check one data value with even parity and one with odd parity. Overall we are faced with a dilemma: forcing the user to decide which cases to check is tedious and potentially error-prone, while doing the analysis for each individual data value is potentially time-consuming. In a BDD-based setting, we have a chance to avoid both of these problems. We will be checking all data values simultaneously. Because of sharing in the BDDs, data values for which the implementation behaves similarly are likely to be collapsed. In essence, the BDDs allow us to do an automatic case analysis to exactly the degree of granularity required in order to ensure soundness.

Figure 5.1 below gives an algorithm for model checking QCTL formulas. The algorithm is expressed in terms of manipulation of relations; these manipulations can be translated into BDD operations in the standard way. In the figure, only the function that determines the set of states satisfying a particular formula is shown; the check to see that every initial state satisfies the given formula is straightforward. The function takes as parameters the (sub)formula to be checked and a list representing the variables which this subformula is in the scope of.

Extending the counterexample generation facility is straightforward. When producing a counterexample for a formula φ at the state s, where the top-level operator of φ is a temporal one, we will have already fixed values for the variables x_0, ..., x_{n-1} on which φ depends. Taking the relation $P(t, x_0, \ldots, x_{n-1})$ that we obtained when evaluating φ, we set the x_i and obtain a relation $Q(t)$ which is the set of states satisfying φ for those values. The x_i will have been chosen so that $Q(s)$ does not hold. We now construct a counterexample for the top-level operator using the standard methods. To show a counterexample for $\forall x\,\varphi$ at the

```
function $check(\varphi, \langle x_0, \ldots, x_{n-1} \rangle)$
if $\varphi = true$
    let $P(s, x_0, \ldots, x_{n-1})$ be identically true
else if $\varphi = (a = x_i)$
    let $P$ be such that $P(s, x_0, \ldots, x_{n-1})$ iff $L(s, a) = x_i$
else if $\varphi = (x_i = d)$
    let $P$ be such that $P(s, x_0, \ldots, x_{n-1})$ iff $x_i = d$
else if $\varphi = \mathbf{AX}\,\psi$
    ...
...
else if $\varphi = \forall x\, \psi$
    $Q := check(\psi, \langle x, x_0, \ldots, x_{n-1} \rangle)$
    let $P(s, x_0, \ldots, x_{n-1})$ iff $\forall x\, Q(s, x, x_0, \ldots, x_{n-1})$
endif
return $P$
```

Figure 5.1: Model checking algorithm for QCTL

state s, we start with the relation $P(t, x, x_0, \ldots, x_{n-1})$ obtained when evaluating φ. Fixing the x_i gives a relation $Q(t, x)$. For some value of x, it must be the case that $\neg Q(s, x)$. We fix x at such a value, display it, and then generate a counterexample for φ.

5.2 Symbolic Abstractions

In this section, we demonstrate that the symbolic manipulation facilities available with BDDs can greatly increase the power of the data abstractions considered in section 4.1. To illustrate the method, we consider verifying the trivial program shown in figure 5.2. (This program is written in the same language used for most of the examples in chapter 4.)

```
input a[8];
output b[8] := 0;

loop
  b := a;
  wait
endloop
```

Figure 5.2: An example program

We wish to show that the next state value of b is always equal to the current state value of a. Using QCTL, we could express this requirement as

$$\mathbf{AG}\,\forall x\,(a = x \rightarrow \mathbf{AX}\,b = x).$$

Let us fix a particular value of x, say 42:

$$\mathbf{AG}(a = 42 \rightarrow \mathbf{AX}\,b = 42).$$

If we wanted to verify just this property, we could use the following abstraction for a and b

$$h(n) = \begin{cases} 0, & \text{if } n = 42; \\ 1, & \text{otherwise.} \end{cases}$$

When we apply this abstraction and compile the program, we obtain the transition relation $R(\hat{a}, \hat{a}', \hat{b}, \hat{b}')$ defined by $\hat{b}' = \hat{a}$. Here, the primes denote next-state variables, and all of the variables range over $\{0,1\}$. Now to check that our program works correctly for the value 42, we would check the following formula at the abstract level:

$$\mathbf{AG}(\hat{a} = 0 \rightarrow \mathbf{AX}\,\hat{b} = 0).$$

The formula would of course turn out to be satisfied. Now we obviously do not want to have to repeat this process for each possible data value.

Suppose now that we were to modify our abstraction function as follows:

$$h_c(n) = \begin{cases} 0, & \text{if } n = c; \\ 1, & \text{otherwise.} \end{cases}$$

We have introduced a new symbolic parameter that our abstraction depends on. Imagine compiling the program with this abstraction; we should get a relation $R(\hat{a}, \hat{a}', \hat{b}, \hat{b}', c)$ that is parameterized by c. Fixing $c = 42$ will give the relation R that we encountered above. If we could run the model checking algorithm on our parameterized relation, we would obtain a parameterized state set representing the states for which our formula is true. Now our specification

$$\mathbf{AG}(\hat{a} = 0 \rightarrow \mathbf{AX}\,\hat{b} = 0)$$

is essentially saying

$$\mathbf{AG}(a = c \rightarrow \mathbf{AX}\, b = c).$$

If the formula turns out to be true for all values of c, we will have proved

$$\forall x\ \mathbf{AG}(a = x \rightarrow \mathbf{AX}\, b = x),$$

which is equivalent to our original specification. The observation now is that by introducing 8 extra BDD variables to encode the possible choices for c, we can in fact:

1. represent h_c with a BDD (the user will supply just h_c);
2. compile with h_c to get a BDD representing $R(\hat{a}, \hat{a}', \hat{b}, \hat{b}', c)$ (the compiler handles this step automatically);

3. perform the model checking to obtain a BDD representing the parameterized state set (the model checker does this automatically); and

4. if necessary, choose a specific c and generate a counterexample (also done by the model checker).

Further note that, in this case, the program behaves identically regardless of the value of $\bar{c}$, so when we compile it, the BDD representing R will be independent of the extra variables that we introduced. As a result, doing the model checking will be no more complex than in the case when we were just verifying

$$\mathbf{AG}(a = 42 \rightarrow \mathbf{AX}\, b = 42).$$

In general, we have found that sharing in the BDDs makes it possible to efficiently perform the parameterized abstraction, compilation, and model checking. We call abstractions such as h_c "symbolic abstractions"; below, we give some more complex examples that make use of these abstractions.

Our first example is a linear sorting array. The array consists of one cell for each integer to be sorted; the program for an individual cell is show in figure 5.3. The cells are numbered consecutively from left to right. In the array, each cell's *left* and *leftsorted* inputs are connected to its left neighbor's y and *sorted* outputs, and each cell's *right* input is connected to its right neighbor's x output. The values to be sorted are the values of the x outputs. The sort proceeds in cycles. During each cycle, exactly half the cells (either all the odd numbered cells or all the even numbered cells) will have their *comparing* output equal to one. These cells compare their own x output with that of their right neighbor. The smaller of these values is placed in y. In addition, if the values were swapped, the cell's *sorted* output is set to zero. During the next clock period, the right neighbor's x and *sorted* values are copied from the first cell's y and *sorted* outputs. When the rightmost cell's *sorted* output becomes one, the sort is complete. In this example, we consider an array for sorting eight numbers. The *comparing* output is set to zero or one depending on the cell's position in the array. The left and right ends of the sorting array are dummy cells for which x is

$2^{16}-1$ and 0 respectively. The left cell's *sorted* output is also fixed at 1.

To verify this program with symbolic abstractions, we used a simple partitioning

$$h_c(n) = \begin{cases} 0, & \text{if } n < c; \\ 1, & \text{if } n \geq c. \end{cases}$$

where c is the parameter. If two numbers are not equivalent according to this abstraction, we can find the truth value of a comparison between them.

The properties which we verified are:

1. for every c, eventually the values of the x outputs are such that all numbers which are less than c come before all numbers which are greater than or equal to c, and this condition holds invariantly from that point on; and

2. for every c, the number of the x outputs which are less than c is invariant except when elements are being swapped.

The first property implies that the array is eventually sorted. The second one implies that the final values of the x outputs form a permutation of the initial values.

We performed the verification by abstracting all the 16 bit variables in the program using the abstraction described above. The temporal formulas corresponding to the two properties are

$$\mathbf{AF}\,\mathbf{AG}((x[7] < c \vee x[6] \geq c) \wedge \cdots \wedge (x[1] < c \vee x[0] \geq c))$$

and, for all j,

$$(\textstyle\sum_{i=0}^{7}(x[i] < c) = j) \rightarrow \mathbf{AG}(\mathit{stable} \rightarrow (\textstyle\sum_{i=0}^{7}(x[i] < c) = j)).$$

To make the formulas more readable, we have written $x[i] < c$ instead of $\hat{x}[i] = 0$ and $x[i] \geq c$ instead of $\hat{x}[i] = 1$. Also, the summation notation is used to denote the number of formulas $x[i] < c$ which are true. Finally, *stable* is an atomic proposition which is true when every cell is at the **wait** statement on line 28. In order to ensure that the cells maintain lockstep, we also checked

$$\mathbf{AG}\,\mathbf{AF}\,\mathit{stable}$$

```
input left[16];
input leftsorted[1];
output sorted[1] := 0;
output comparing[1] := <0 or 1>;
output swap[1] := 0;
output x[16];
output y[16];
input right[16];

loop
  if comparing = 1
    swap := (x < right);
    wait;
    if swap = 1
      y := x;
      x := right;
      sorted := 0
    else
      y := right
    endif;
    wait
  else
    wait;
    wait;
    x := left;
    sorted := leftsorted
  endif;
  comparing := !comparing;
  wait
endloop
```

Figure 5.3: A sorting cell program

Verifying these properties required just under five minutes of CPU time on a Sun 4. In addition, checking these properties on the unabstracted program was not feasible due to space limitations.

5.3 Symbolic Compositions

For our last example, a pipelined arithmetic circuit, we will use symbolic abstractions together with an additional technique called "symbolic compositions". Suppose that we have a system with a number of related processes $M_0, \ldots, M_{n-1}$. Also suppose that we wish to verify a class of properties $\varphi_0, \ldots, \varphi_{n-1}$, where φ_i describes the interaction of M_i with the remainder of the system, modeled by M. Using the compositional reasoning ideas described in chapter 3, we might try to check φ_i on just the composition $M \parallel M_i$. Instead of doing this for each individual i however, we may be able to use symbolic parameters to do all of the checks at once. To see how this might be possible, we consider a simple example.

Let $M_0, \ldots, M_{15}$ be registers, where M_i is described by the program in figure 5.4. Note that i is used as a parameter in this program. During each cycle, one of the registers is set to value of the input a. Each register will have a different output $b[i]$.

```
input addr[4];
input a[16];
output b[16]

loop
  if addr = <i>
    b := a
  endif;
  wait
endloop
```

Figure 5.4: An example program

Suppose that we want to verify the following class of properties:

$$\mathbf{AG}\,\forall x\,(addr = i \wedge a = x \rightarrow \mathbf{AX}(b[i] = x)),$$

where i ranges from 0 to 15. To verify the property for $i = 7$, it will obviously be enough to check

$$\mathbf{AG}\,\forall x\,(addr = 7 \wedge a = x \rightarrow \mathbf{AX}(b[7] = x))$$

on M_7. Now suppose we rename $b[7]$ to b in M_7 and in the above property before doing the check. This obviously will not affect whether the verification succeeds or not. Now consider how we can do this for all i simultaneously. Taking the program for M_i, compiling it, and renaming $b[i]$ to b can be done by just compiling the program above using a new 4 bit symbolic parameter to represent i. The result is a parametric representation of M_i . Using that same symbolic parameter, we can express the class of properties (after renaming) with the formula

$$\mathbf{AG}\,\forall x\,(addr = i \wedge a = x \rightarrow \mathbf{AX}(b = x)).$$

When we run the model checker now, the effect is to check the specification involving M_i using just M_i. For this particular example, the whole verification can be done using about the same amount of time and space as would be required for checking just one of the properties. Note that when doing the verification, we have managed to avoid composing all of the M_i, and hence we never deal with more than a small part of the system state space.

We now turn to a more extensive example, a pipelined arithmetic unit. A block diagram circuit is shown in figure 5.5. This example was first described by Burch *et al.* [23]. It performs three-address arithmetic and logical operations on operands stored in a register file. The pipeline operates as follows:

1. During the first cycle of the instruction, operands are read from the register file into the instruction operand registers.

2. During the second cycle, the result of the operation is computed and stored in the pipeline register after the ALU.

3. In the third, the result is written back to the register file.

Thus, performing an operation requires three cycles. Each instruction to the pipeline specifies the source and destination registers and the operation to perform. In addition, the pipeline has a *stall* input that indicates that the instruction is invalid and should be ignored. More specifically, the instruction's destination register should not be affected if the *stall* input is true. The *stall* signal might, for example, be used to indicate an instruction cache miss; the signal would be asserted until an instruction is fetched from main memory. In order to allow results to be used before they are actually written into the register file, data can be fed from the ALU output or from the ALU output register back to the ALU operand registers. To simplify matters slightly, we shall consider a pipeline that only performs addition operations, but the same techniques can be used to verify other operations as well.

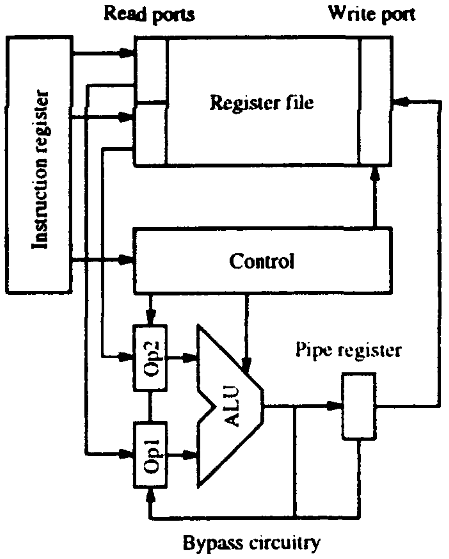


Figure 5.5: Pipeline circuit block diagram

The specification that we will check is the following: for every possible value of the source and destination addresses, the value of the destination register in three cycles will be equal to the sum of the values in the source registers in two cycles. The use of the values in the source registers two cycles hence is necessary to allow for the possibility that those registers may be in the process of being updated.

$$\begin{aligned}&\mathbf{AG}(\mathit{srcaddr1} = i \wedge \mathit{srcaddr2} = j \wedge \mathit{destaddr} = k \wedge \neg \mathit{stall}\\ &\qquad \rightarrow \forall a \forall b\ \mathbf{AX}\,\mathbf{AX}((\mathit{reg}[i] = a) \wedge (\mathit{reg}[j] = b)\\ &\qquad\qquad \rightarrow \mathbf{AX}(\mathit{reg}[k] = a + b)))\end{aligned}$$

Also, any given register is not affected if either it is not the destination register or the current instruction is stalled.

$$\begin{aligned}&\mathbf{AG}(\mathit{stall} \vee \mathit{destaddr} \neq i\\ &\qquad \rightarrow \forall a\ \mathbf{AX}\,\mathbf{AX}((\mathit{reg}[i] = a) \rightarrow \mathbf{AX}(\mathit{reg}[i] = a))).\end{aligned}$$

Observe that to verify one of these properties, we should need only the registers involved ($\mathit{reg}[i]$, $\mathit{reg}[j]$, and $\mathit{reg}[k]$) plus the other parts of the pipeline. Thus, we are in a position to use a symbolic composition. We introduce new symbolic parameters i, j and k. We then compile three copies of the program for a register, parameterized by i, j and k, with the *reg* output renamed to *regi*, *regj* and *regk* respectively. During the compilation process, we also want to abstract the data values that can be stored in the registers. For verifying the addition operation, we will introduce symbolic parameters a and b, and then use the abstraction

$$h_{a,b}(n) = \begin{cases} 0, & \text{if } n = a; \\ 1, & \text{if } n = b; \\ 2, & \text{if } n = a + b; \\ 3, & \text{otherwise.} \end{cases}$$

Now we would like to check

$$\begin{aligned}&\mathbf{AG}(\mathit{srcaddr1} = i \wedge \mathit{srcaddr2} = j \wedge \mathit{destaddr} = k \wedge \neg \mathit{stall}\\ &\qquad \rightarrow \mathbf{AX}\,\mathbf{AX}((\widehat{\mathit{regi}} = 0) \wedge (\widehat{\mathit{regj}} = 1)\\ &\qquad\qquad \rightarrow \mathbf{AX}(\widehat{\mathit{regk}} = 2)))\end{aligned}$$

and

$$\mathbf{AG}(\mathit{stall} \vee \mathit{destaddr} \neq i \rightarrow \mathbf{AX}\,\mathbf{AX}((\widehat{\mathit{regi}} = 0) \rightarrow \mathbf{AX}(\widehat{\mathit{regi}} = 0))).$$

There is one minor problem however: the map $h_{a,b}$ may not be well-defined. Suppose, for example, that $a = b$; then $h_{a,b}(a)$ could be 0 or 1.

Further, if $a = b = 0$, it could even be 2. We can resolve this difficulty in one of two ways. The first is to do a by-hand case analysis in order to get a series of well-defined maps. For this example, we could look at the following cases:

1. The possible abstract values are $a \neq 0$, 0, and everything else; we check that the system works correctly when both operands are 0, and when one operand is 0 and the other is a.

2. The possible abstract values are $a \neq 0$, $a + a$, and everything else. We verify that the pipeline works when both operands are a.

3. The possible values are a, b, $a+b$, and everything else. We require $a \neq b$, and for both a and b to be nonzero. Then we check that the system is correct when the operands are a and b.

It is easy to see that this covers all possibilities, and in each case we can build a well-defined abstraction mapping. Note that with this method, we encode a *set of k abstract values using* $\lceil \log_2 k \rceil$ bits. These second way to fix the problem is to allow the abstract classes to overlap, and to encode the k possible abstract values with k bits. In the case of $h_{a,b}$ above, we would use three bits, for a, b and $a + b$, and have

$$h_{a,b}(n) = \begin{cases} 0, & \text{if } n \neq a \wedge n \neq b \wedge n \neq a + b; \\ 1, & \text{if } n = a \wedge n \neq b \wedge n \neq a + b; \\ 2, & \text{if } n \neq a \wedge n = b \wedge n \neq a + b; \\ 3, & \text{if } n = a \wedge n = b \wedge n \neq a + b; \\ 4, & \text{if } n \neq a \wedge n \neq b \wedge n = a + b; \\ 5, & \text{if } n = a \wedge n \neq b \wedge n = a + b; \\ 6, & \text{if } n \neq a \wedge n = b \wedge n = a + b; \\ 7, & \text{if } n = a \wedge n = b \wedge n = a + b. \end{cases}$$

Then, to say that *regi* has the value a, we would write

$$\widehat{regi} \in \{1, 3, 5, 7\}.$$

We used the second method for this example.

The largest pipeline example we tried had 64 registers in the register file and each register was 64 bits wide. This circuit has more than 4,000 state bits and nearly 10^{1300} reachable states. The verification required less than 25 minutes of CPU time on a Sun 3/60. The verification time scales polylogarithmically in the number of registers and linearly in the width of registers. Burch, Clarke, and Long [22] verified essentially the same circuit using no abstraction. With 8 registers, each 32 bits wide, they required 4 hours and 20 minutes of CPU time on a Sun 4 to complete the verification. In addition, their verification times were growing cubicly with the number of registers and quadratically with the register width.

Chapter 6

Verification of the Futurebus+ Cache Coherence Protocol

In this chapter, we apply some of the ideas from chapters 2 through 5 to the verification of the cache coherence protocol described in the IEEE Futurebus+ standard. Our goal is to demonstrate that the methods can be used to verify designs of realistic complexity. The work described below is an extension of work that we reported earlier [29].

6.1 Overview of the Protocol

Futurebus+ is an emerging bus standard for high-performance multiprocessors. The goal of the committee that developed Futurebus+ was to create a public standard for bus protocols that was unconstrained by the characteristics of any particular processor or device technology and that would be widely accepted and implemented by vendors. It has been adopted by the Navy's next-generation computer resources program as its standard linear backplane, and companies such as DEC, Sun, Motorola and Force Computers are developing Futurebus+ products. The Futurebus+ specification is actually a number of standards, covering issues from physical interconnection through high-level protocols. We will be concerned with the *IEEE Standard for Futurebus+*—

Logical Protocol Specification (IEEE Standard 896.1–1991) [59]. Part of this standard is a *cache coherence protocol* designed to insure consistency of data in systems composed of many processors and caches interconnected by multiple bus segments. (For an overview of a number of cache coherence protocols, see the article by Archibald and Baer [3].)

Consider a multiprocessor system such as the one shown in figure 6.1. Each of the processors P1, P2, and P3 has access to a central *shared memory*, M. P3 is on the same bus as M, so read and write requests from P3 can be delivered to M directly. In contrast, requests from P1 and P2 must pass through a communications network before reaching M. (There may actually be many processors and memories scattered throughout the system, but each memory location must belong to a single home memory. Also, all of processors that can access the memory location must form a tree rooted at the memory.) There are two main problems that arise in accessing memory.

1. When there are many processors contending for access to M, the *bandwidth* required to ensure adequate performance can be very high.

2. The *latency* of servicing requests that must pass through the network can be very long.

In order to alleviate these problems, each processor is equipped with a *cache*. A cache can hold copies of some of the memory locations in M. When a processor wants to read or write, it can often obtain the data from its cache, or store it in the cache. This is a fast operation, and because programs exhibit *locality of reference*, a piece of data is typically moved into a cache once and then accessed a number of times. However, while caching is effective for reducing latency and bandwidth requirements, it can destroy the original shared memory semantics of accesses. Suppose, for example, P1 obtains a copy of some memory location in its cache and then writes to that location. If P3 now wants to read the same location, it must somehow know that the data is stored in P1's cache, and that the copy in memory is out of date. Maintaining shared memory semantics is the purpose of the cache coherence protocol.

In the Futurebus+ protocol, sequences of consecutive memory locations are grouped together into *cache lines*. Each cache line is treated as

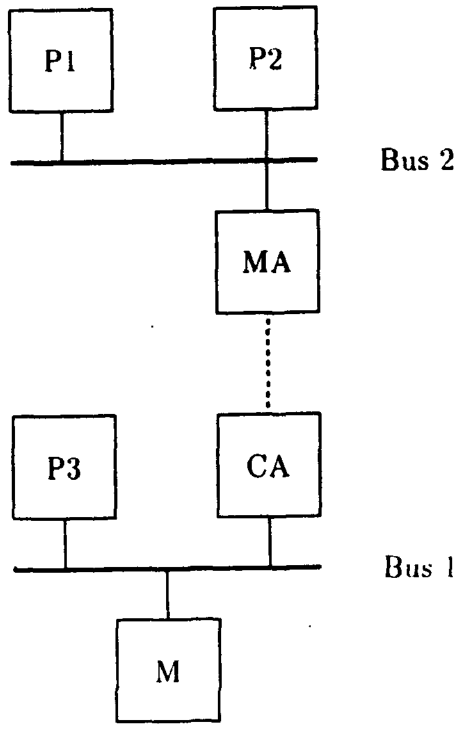


Figure 6.1: Multiprocessor system

a unit for coherence purposes. Under the protocol, coherence is maintained on individual buses by having the individual processors *snoop*, or observe, all bus transactions. As an example, consider figure 6.1 again. Suppose that P1 obtains a copy of a cache line and writes to one of the locations in the line. Then P2 tries to read a location in the same line by putting a read request on the bus. P1 will snoop the read request and will *intervene* to supply the data directly to P2. Coherence across buses is maintained using special *cache agents* and *memory agents* (CA and MA in the figure). The CA/MA pair is collectively called a *bus bridge*. The cache agent is responsible for issuing commands on bus 1 on behalf of the remote processors P1 and P2. Similarly, the memory agent is responsible for representing the memory M on bus 2. If P1 issues a read on bus 2, then MA will pass the request down to the cache agent CA, and CA will reissue the read on bus 1. Next, the memory supplies the data to CA, and CA passes it back to MA, which in turn forwards it to P1. Obviously, a sequence such as this can tie up the buses for quite a while. Thus, in order to increase performance, the protocol uses *split transactions*. When a transaction is split, it is divided up into separate initiation and completion phases. In our example, the read that P1 issues would be split to free up bus 2. While the read request is propagating towards memory, bus 2 can be used by P2 to issue other requests. When MA finally receives the data that P1 requested, it issues an explicit *response* transaction to supply the data.

There are two other performance optimizations used in the protocol. First, writes are not propagated back to main memory immediately. Instead, the data from the write is simply stored in the cache. Later, when the line needs to be replaced in the cache, an explicit *copyback* is used to return the up-to-date data to main memory. Second, processors may obtain data from other processors' transactions by *snarfing*. Suppose for example, that P1 and P2 both wish to obtain readable copies of some cache line. They arbitrate for the bus, and let us suppose that P2 wins the arbitration and issues the read request. The memory agent splits the transaction, goes off and obtains the data, and then issues a response on bus 2. P2 will take the data from this response, but P1 is also allowed to obtain the data as it passes on the bus. When this happens, both P1 and P2 end up with valid copies of the line.

The Futurebus+ protocol falls in the class of *MESI* coherence protocols. MESI stands for "Modified-Exclusive-Shared-Invalid" and represents the possible states that a cache line can be in within a given cache.

1. A cache that has no information about a particular cache line is in the *invalid* state for that line. Obviously, neither read nor write access is allowed to any of the memory locations within the line.

2. A cache that is in the *shared-unmodified* state has a readable copy of the cache line, and other caches may have copies as well. Writing is not allowed when the cache line is in this state.

3. A cache that is in the *exclusive-modified* state has a readable and writable copy of the line. It is the only place in the system where up-to-date data is stored, and hence must supply the data when someone else issues a read request.

4. The last state, *exclusive-unmodified*, represents a combination of the *shared-unmodified* and *exclusive-modified* states. In this state, the cache has a copy of the data and only reading is allowed. However, it is also guaranteed that no other cache has a copy of the data. If the processor whose cache has the *exclusive-unmodified* copy decides to write to a location in the cache line, the line is simply placed in the *exclusive-modified* state and the write proceeds. There is no need to issue any sort of transaction to eliminate copies that may be in other caches. On the other hand, if the processor never writes to the line and it is necessary to purge the line from the cache, then there is no need to copy the data back to main memory.

Next, we describe the different types of bus transactions that devices can issue. There are two basic read transactions: *read-shared* and *read-modified*. The former is used to request a readable copy of a cache line, while the latter requests both read and write access. Both types of transactions may be split. In the case of a *read-shared* transaction, other devices are allowed to snarf that data as it is supplied to the requester. Note that the *read-modified* transaction requires any copies of the cache line in other caches to be eliminated. Because of this, *read-modified* transactions can be split for two distinct reasons:

1. the supplier of the cache line may split the transaction if it is not able to immediately respond with the data (*splitting for access*); and

2. a processor or cache agent may split the transaction if it currently has a copy of the line and cannot invalidate that copy immediately (*splitting for invalidation*).

Snarfing is obviously not allowed on *read-modified* transactions.

If a cache currently has a *shared-unmodified* copy of a cache line, it may request write access by issuing an *invalidate* command. This transaction causes other caches with shared copies to eliminate those copies. Any of these caches may delay the invalidation process by splitting the *invalidate* transaction. Once a cache has obtained an *exclusive-modified* copy of a cache line, it is the sole holder of the data in that line. As such, it is responsible for intervening in any read requests by other caches. By intervening, it supplies the data to someone else, and hence transitions out of the *exclusive-modified* state. The only other way that it can exit this state is by issuing a *copyback* transaction to return the data to main memory. During a *copyback*, any cache that would like to obtain a copy of the data in the line is allowed to snarf it. This includes the cache issuing the *copyback*.

There are also two basic types of responses, corresponding to the two types of reads. A *shared-response* is used to supply data to a cache whose earlier *read-shared* was split. Other devices may snarf data from the *shared-response*. A *modified-response* is used to grant read-write access, and as such it is issued in response to split *read-modified* and *invalidate* transactions. Recall that a *read-modified* can be split either for access or for invalidation. Because of this, there are actually two forms of *modified-response*: one supplying data, and one that is used only as an acknowledgment of invalidation (an *address-only modified-response*). Note also that a single *invalidate* may be split by multiple devices. Hence there must be some way to tell when all of them have finished invalidating. This is done by allowing *modified-response* transactions to be split. Suppose, for example, that P1 and P2 are both invalidating. P1 finishes and issues a ***modified-response***. P2, which is still invalidating, cannot let this response pass, so it splits. Later, when P2 finishes invalidating, it issues a second *modified-response*. Since P2 is the last device done, this response is not split. The cache that issued the original ***invalidate*** proceeds when it sees this unsplit *modified-response*. Similarly, ***read-modified*** transactions may be split by multiple devices, and as with ***invalidate***, an unsplit *modified-response* signals the requesting device that it may proceed.

Devices communicate their requests to split transactions, snarf data, or intervene using three bus lines called *SR*, *TF*, and *IV*. These are wire-or signals: effectively, each device i has outputs sr_i, tf_i, and iv_i, and $SR = \bigvee_i sr_i$, etc. (Thus, if any device requests that a transaction be split, it will be split.) A device asserts sr_i to request that the current transaction be split. It raises tf_i when it wants to snarf data from the current transaction. Finally, if it observes a read request, and it has an *exclusive-modified* copy of the requested cache line, it asserts iv_i to indicate that it will supply the data for the read.

Example 6.1 We consider a sequence of transactions dealing with some fixed cache line for the system shown in figure 6.1. Initially, all caches have *invalid* copies of the cache line. If P1 wants a readable copy of the cache line, it issues a ***read-shared*** on bus 2. The memory agent MA cannot supply the requested data immediately, so it asserts its *sr* output to split the transaction. It passes the request to CA, which issues the ***read-shared*** on bus 1. The memory supplies the data to the cache agent, and during the transfer, P3 asserts its *tf* output and snarfs the data. P3 now has a *shared-unmodified* copy of the cache line. The data is passed back to MA, and MA issues a ***shared-response*** to provide the data to P1. P2 snarfs the data by asserting *tf* during the response, and both P1 and P2 wind up with *shared-unmodified* copies. P1 now requests write access by issuing an ***invalidate*** transaction. P2 asserts *sr* to split the transaction for invalidation, as does MA. P2 finishes invalidating and issues a ***modified-response***. Since P3 is not yet invalid, the memory agent must split this ***modified-response***. The request for invalidation propagates to CA, which issues ***invalidate*** on bus 1. P3 invalidates immediately, and CA informs MA of this. The memory agent issues a *modified-response* which is not split, and P1 transitions to the *exclusive-modified* state. P2 now requests read and write access by issuing a *read-modified*. P1 intervenes by asserting its *iv* output and supplies the data to P2. P1 transitions to the *invalid* state, while P2 becomes *exclusive-modified*. P2 decides to kick the line out of its cache, so it issues a *copyback* to return the data to memory. The memory agent MA picks up the data as P2 goes to *invalid*. The data is passed to CA, which issues the *copyback* on bus 1. P3 now requests read access and issues a *read-shared*. If CA does not snarf the data by asserting *tf*, then P3 transitions to the *exclusive-unmodified* state. Later, if P3 decides to write, it goes immediately to *exclusive-modified*, updates the line, and then issues a *copyback* to return the data to M. □

Split transactions are controlled using *requester and responder attributes*. When a device issues a request that is split, it acquires a requester attribute that indicates the type of response it expects to receive. The device that splits the request gets a responder attribute that tells what type of response it will eventually issue. The possible requester attributes are as follows:

1. A cache has the *requester-shared* attribute when it is waiting for a *shared-response*.

2. The *requester-exclusive* attribute is true when a device is waiting for a *modified-response*.

3. The final attribute, *requester-waiting*, will be discussed below.

The responder attributes are similar:

1. A device has the *responder-shared* attribute when it must eventually issue a *shared-response*.

2. The *responder-exclusive* attribute is used to indicate that the cache must eventually issue a *modified-response* to supply data.

3. When a processor has the *responder-invalidate* attribute, it must issue an address-only *modified-response* to signal the completion of invalidation.

Each cache line has separate requester and responder attributes.

Under the protocol, there may be only one pending transaction per cache line per bus in the system. Hence, when a transaction is split, there must be a way of preventing other transactions for the same cache line from proceeding. This is done using another bus line, *WT*. *WT* is also a wire-or signal, so any device may drive *WT* high by setting its individual *wt* output. If a module has any of the responder attributes discussed above, then it is already processing one transaction for the same cache line. When it observes another transaction for the cache line, it asserts *wt* to abort this new request. A device that tries to issue a transaction and observes *WT* acquires the *requester-waiting* attribute. It keeps this attribute until it sees a *shared-response* or an unsplit *modified-response* for the same cache line. At that point, it may retry its original request.

There is one slight exception to the rule of one pending transaction per cache line per bus. Consider the system of figure 6.1, and suppose that P1 and P3 both have *shared-unmodified* copies of some cache line. Now assume that both processors decide to write to the cache line at roughly the same time and both issue *invalidate* transactions. The cache agent CA must split the *invalidate* on bus 1 since P1 has a copy of the cache line. Similarly, MA has to split P1's *invalidate*. At this point, we have a conflict: P1 is trying to invalidate P3 and P3 is trying to invalidate P1. In the protocol, this *invalidate-invalidate collision* is resolved by allowing an *invalidate* to be issued underneath an already pending *invalidate*. First priority is given to the invalidate that is proceeding away from main memory. Thus, MA will issue an invalidate to eliminate the data in P1's cache. After that, CA issues an address-only *modified-response* to give P3 exclusive access. Then the cache agent uses a *read-modified* to get the updated data from P3, and the data is passed to MA. MA issues a *modified-response* to give an *exclusive-modified* copy of the cache line to P1.

The IEEE Standard for Futurebus+—Logical Protocol Specification [59] contains two sections dealing with the cache coherence protocol. The first, a description section, is written in English and contains an informal and readable overview of how the protocol operates, but it does not cover all scenarios. The second, a specification section, is intended to be the real standard. This section is written using *attributes*.

An attribute is essentially a boolean variable together with some rules for setting and clearing it. The attributes are more precise, but they are difficult to read. The behavior of an individual cache or memory is given in terms of roughly 300 attributes, of which about 45 deal specifically with cache coherence. (These 45 attributes reference many of the other attributes as well.) As an example, the following attribute for cache modules tells when the cache has a ***shared-unmodified*** copy of a particular cache line:

> **SHARED_UNMODIFIED**. A *CACHE* or *CACHE_AGENT* shall set *SHARED_UNMODIFIED* and clear *INVALID* ∨ *EXCLUSIVE_UNMODIFIED* ∨ *EXCLUSIVE_MODIFIED* if *MASTER* ∧ (*INVALID_STATUS* ∧ ¬*ADDRESS_ONLY* ∧ (*READ_SHARED* ∨ *READ_MODIFIED*) ∨ *KEEP_COPY* ∧ (*COPY_BACK* ∨ *SHARED_RESPONSE*)) ∨ *CACHED* ∧ (*REQUESTER_SHARED* ∧ *SHARED_RESPONSE* ∧ *INVALID_STATUS* ∧ ¬*ADDRESS_ONLY* ∧ *TRANSACTION_FLAG_STATUS* ∨ *SNARF_DATA* ∧ ¬*ADDRESS_ONLY* ∨ *REQUESTER_EXCLUSIVE* ∧ *MODIFIED_RESPONSE* ∧ ¬*ADDRESS_ONLY* ∧ *SPLIT_SATUS* ∨ ¬*INVALID_STATUS* ∧ *KEEP_COPY* ∧ (*READ_SHARED* ∨ *READ_INVALID*)).
>
> A *CACHE* or *CACHE_AGENT* may set *SHARED_UNMODIFIED* and clear *EXCLUSIVE_UNMODIFIED* if *EXCLUSIVE_UNMODIFIED*.
>
> A *CACHE* or *CACHE_AGENT* shall not allow modify access to the data in a cache line if *SHARED_UNMODIFIED* is set. A *CACHE* or *CACHE_AGENT* may allow read access to the data in a cache line if *SHARED_UNMODIFIED* is set.

Note that even in the specification section, some aspects of a module's allowed behavior are described informally. For example, the above attribute specifies a processor's read-write permissions in English. Further, the bus bridge operation is not completely specified in either section. We are given only that externally, cache agents and memory agents should "look like" caches and memories. There are some examples of bus bridge operation and a description of the collision resolution mechanism, but the coordination between cache agent and memory agent is not specified in detail. A major part of our verification effort was devoted to making an appropriate model of the bus bridges.

6.2 Modeling the Protocol

Clearly, verifying a fully detailed model of the protocol at the level of the attributes would not be practical. Even if the attributes were completely precise and covered all aspects of the allowed behavior, the verification tools would not be able to handle this model. Further, since the attributes are very difficult to understand, it would not be easy to make appropriate abstractions to simplify the verification process. For these reasons, we used the English language description as the basis for our model. Situations where this description was ambiguous or incomplete were resolved by referring to the attributes. While constructing the model, we made a number of simplifications and abstractions (listed below). For each abstraction, we describe how it would be justified using the techniques discussed previously.

1. The standard specifies how modules should respond to exceptional situations, such as detection of a parity error during a data transfer. In our model, we assumed that these cases do not occur. Similarly, the standard describes power-up, reset, and configuration protocols. We modeled only the case of steady-state operation.

2. A fairly complex protocol is used to arbitrate for the bus and issue a transaction. In our model, a complete arbitration/transaction cycle is modeled as a single state transition. Given an actual implementation, we would use abstraction via observers to make this type of simplification. Our observer processes would watch the low-level handshaking and output in one step the high-level indication of which module was selected as master and which transaction it issued. This is similar to the abstraction of a pipelined

system in example 4.9.

3. We modeled only the transactions involving one cache line. This type of simplification can be justified using abstraction via observers and symbolic parameters. Suppose that we have an implementation in which the cache line under consideration is the one beginning at address a. We also assume that this cache line, plus its associated tag bits and attributes, is stored at some location b in the cache RAM, where b depends on a. We describe the relevant part of the cache as a symbolic composition. Our observer process then looks at the location b to determine whether the cache line is in fact in the cache, and if so, what its state is. The observer outputs this state at the abstract level, or outputs *invalid* if the line is not stored in the cache at location b.

4. The data in the cache line is modeled as a single bit instead of 64 bytes. We can use a symbolic abstraction to perform this abstraction. The bit can be thought of as representing whether the value in the line is the 64 byte value c, or whether it is some other value.

5. Components such as processors nondeterministically issue reads and writes to the selected cache line. To justify this abstraction, we simply hide the internal state of the processor using the restriction operator and then apply *collapse* to reduce the state space. (In fact, the processor model is essentially $\top$, so we know it can simulate whatever the real processor would do.)

6. Responses to split transactions are issued after arbitrary delays. This would be justified in essentially the same way as the previous abstraction.

7. The bus bridge model is highly abstracted. This model is discussed in detail below.

8. The standard specifies some types of transactions that are intended mainly for peripheral devices doing I/O. Cache coherence is generally not maintained when these instructions are used, so

we assumed that they would not be issued for the cache line that we modeled.

In our model, all of the devices on a single bus are composed synchronously, i.e., all of them update their state during a transaction on that bus. Different buses are composed in an asynchronous manner: during a step of the system, the components on one bus execute a transaction, while those on other buses make idle transitions.

In our model of a bus bridge, the cache agent and memory agent share a small amount of internal state. The set of possible internal states represents a generalization of the possible states of a cache line in a processor cache. These states are as follows:

1. When the bridge is in the *invalid* state, it has no information about the cache line.

2. In the *local-shared* state, the bridge has an internal copy of the cache line, and other caches below (on the cache agent side of) the bridge may have copies. This bus bridge state corresponds to *shared-unmodified* in a processor cache.

3. In the *shared-valid* state, the bridge has an internal copy, and caches both above and below the bridge may have copies. This also corresponds to *shared-unmodified*.

4. The *shared-invalid* state is a situation in which the bridge does not have a copy of the line, but caches both above and below the bridge may. (Note that while bridges must maintain cache tags for the lines that are in remote caches, they need not store the line itself.) As with the previous two states, this one corresponds to *shared-unmodified* in a processor cache.

5. The bridge may be in the *remote-shared-unmodified-valid* state, indicating that the bridge has a copy of the line, and that caches above the bridge may also have copies. This corresponds to the *exclusive-unmodified* state in a processor cache.

6. The *remote-shared-unmodified-invalid* state is similar to the previous state, but the bridge does not have a copy.

7. In the *exclusive-unmodified* state, the bridge has an unmodified copy of the cache line, but no other caches in the system may have copies. This is also analogous to the *exclusive-unmodified* processor cache state.

8. The *remote-exclusive-modified* state in the bridge means that some cache above the bridge has an *exclusive-modified* copy of the line.

9. The *remote-shared-modified* state is one where the bridge has a copy of the line, remote caches may have copies, and the data is different than that stored in main memory. Like the previous state, this one corresponds to the *exclusive-modified* state of a cache line.

10. Finally, the *exclusive-modified* bridge state corresponds to the *exclusive-modified* processor cache state, and represents a situation where the bridge has the only valid copy of the data in the line.

In our initial model, the cache agents and memory agents chose commands nondeterministically based only on the internal bridge state. There was no explicit passing of commands between the two agents. Consider, for example, a configuration like the one of figure 6.1. Suppose that all of the caches and bridge are in the *invalid* state. If P1 issues a *read-shared*, then the MA will examine the bridge state, find that it is *invalid*, and decide that it must split the read request. At some later point, CA can examine the bridge state, see that it is *invalid*, and nondeterministically choose to issue a *read-shared* on bus 1. Suppose that this read completes and that P3 snarfs the data: then the bridge transitions to the *local-shared* state. Later still, the memory agent may get a chance to execute. Seeing that the bridge is in the *local-shared* state and that it owes a response to P1, it may nondeterministically issue a *shared-response*. If this happens, then P1 gets the data and the bridge transitions to either the *shared-valid* or the *shared-invalid* state. Note that with this model, there is no guarantee of progress. The advantage is that the bridge model is relatively simple, which helps make the verification possible. Further, it can simulate a wide variety of possible implementations. For example, a bridge that detected sequential accesses and attempted to prefetch cache lines would be covered by this model. The abstractions used in constructing the model can be justified using abstraction via observers plus hiding and collapsing.

The protocol model was written in the hardware description language used by SMV. SMV ("Symbolic Model Verifier") is a BDD-based CTL model checker developed by McMillan as part of his thesis [67]. There, he used SMV to verify another hierarchical cache coherence protocol, the protocol used by the Encore Gigamax [68, 67]. Due to the size of the model (about 3000 lines of code), we will not give it here. However, in order to give a feel for the language, a simplied fragment is shown in figure 6.2. This fragment deals with the responder attribute for the cache line being modeled and the *wt* output of a device. The language provides module facilities for structuring designs (line 1). The `VAR` declaration (line 2) specifies state components. All components have finite type: in this case, we declare a boolean and a value with enumerate type. The way components change is specified using the `ASSIGN` declaration (line 5). We can specify either the initial and next state values of the component (line 6), or we can say that the component is invariantly equal to some expression (line 24). Components without assignments are treated as inputs. The language includes facilities for specifying nondeterminism; by assigning a set to a component (line 26), we indicate that the value of the component should be chosen from the elements of the set. The model consists of four major modules, representing processor caches, memories, cache agents, and memory agents. There are smaller modules defining pieces such as the buses. Each module is essentially a series of **case** statements, one per component. This **case** statement tells how the component changes based on the current cache line state, requester and responder attributes, bus master, command, etc.

6.3 Specifying Cache Coherence

In this section, we discuss the specifications used in verifying the protocol. More exhaustive specifications are possible; for example, we might

```
MODULE responding-device

VAR
wt: boolean;
responder: {none, exclusive, invalidate, shared};

ASSIGN
init(responder) := none;
next(responder) :=
  case
  WT: responder;
  master:
    case
    CMD=shared-response & responder=shared: none;
    CMD=modified-response &
      responder in {invalidate, exclusive}: none;
    1: responder;
    esac;
  CMD=read-shared & sr: shared;
  CMD=read-modified & sr: exclusive;
  CMD=invalidate & sr: invalidate;
  CMD=modified-response & !sr &
    responder=invalidate: none;
  1: responder;
  esac;

wt :=
  case
  WT: {0, 1};
  !master & !(responder=none) &
    !(CMD in {shared-response, modified-response}): 1;
  1: 0;
  esac;
```

Figure 6.2: A small part of the program describing the protocol

develop specifications of each individual type of device describing how it responds to different transactions. Here, we have only tried to describe what cache coherence is, not how it is achieved. We begin with some basic safety properties. Each device model includes two flags, *bus-error* and *error* that become true when the device observes an illegal combination of bus signals or an unexpected transaction. These conditions are defined in the standard. For example, while devices may assert *sr* during an *invalidate* transaction, they should never assert *iv*. If a module observes *IV* high during an *invalidate*, the *bus-error* state component becomes 1. The *error* flag becomes true when a device observes a transaction which should not occur given its internal state. For example, if a processor cache has a *shared-unmodified* copy of a cache line, and a *read-shared* is issued, then no other cache should intervene (by asserting *iv*) in that transaction. If another cache does intervene, then that cache must have an *exclusive-modified* copy of the line. This should not be the case since the first cache has a readable copy. Thus, we have the following formula for every device d in our system:

$$\mathbf{AG}(\neg d.\mathit{bus\text{-}error} \wedge \neg d.\mathit{error}). \qquad (6.1)$$

Here, $d.\mathit{error}$ indicates the *error* state component in device d. We also require that if the processor cache P1 has an exclusive copy of the cache line, then no other cache P2 should have a copy.

$$\mathbf{AG}(P1.\mathit{exclusive} \rightarrow P2.\mathit{state} = \mathit{invalid}) \qquad (6.2)$$

Here, $P1.\mathit{exclusive}$ is an abbreviation for

$$P1.\mathit{state} \in \{\mathit{exclusive\text{-}unmodified}, \mathit{exclusive\text{-}modified}\}.$$

The next two properties state that data must be consistent within the caches: if two caches have readable copies, then they must agree on the data. Similarly, if a cache has a copy and memory is up-to-date, then the data in the cache and the data in memory must be the same.

$$\mathbf{AG}(P1.\mathit{state} = \mathit{shared\text{-}unmodified} \wedge P2.\mathit{state} = \mathit{shared\text{-}unmodified} \rightarrow P1.\mathit{data} = P2.\mathit{data}) \qquad (6.3)$$

$$\mathbf{AG}(P1.\mathit{unmodified} \wedge \neg M.\mathit{memory\text{-}line\text{-}modified} \rightarrow P1.\mathit{data} = M.\mathit{data}) \qquad (6.4)$$

The abbreviation *P1.unmodified* means

$$P1.state \in \{\textit{shared-unmodified}, \textit{exclusive-unmodified}\}.$$

The *memory-line-modified* component of a memory is false when the data in memory is supposed to be accurate.

Our final safety property is one specifying *strong sequential consistency* or *strong coherence.* This property states that caches must always read up-to-date data (i.e., the last value written).

$$\begin{aligned}\forall a\ \mathbf{AG}(&P1.state = \textit{exclusive-modified} \land P1.data = a \\ &\rightarrow \mathbf{AX}\,\mathbf{A}(\textit{write}\ \mathbf{V}\ P2.unmodified \rightarrow P2.data = a))\end{aligned} \qquad (6.5)$$

The formula *write* is true whenever one of the processor caches is in the *exclusive-modified* state, i.e., when one of them can write the data in the line.

We would also like to check that the protocol ensures some form of progress. However, our initial model does not have this property. We can state an absence-of-deadlock property, i.e., that it is always *possible* for a cache to get readable and writable copies of the line.

$$\mathbf{AG}\,\mathbf{EF}\ P1.state = \textit{shared-unmodified} \qquad (6.6)$$
$$\mathbf{AG}\,\mathbf{EF}\ P1.state = \textit{exclusive-unmodified} \qquad (6.7)$$
$$\mathbf{AG}\,\mathbf{EF}\ P1.state = \textit{exclusive-modified} \qquad (6.8)$$

Unfortunately, these are not ACTL properties, and hence checking that they hold for the model does not guarantee that they are true in an actual system. We can use them for debugging purposes though; if one of these properties is false, then we can examine the counterexample produced by the model checker to see whether it represents a real deadlock. This is the approach we originally took. In section 6.6, we discuss strengthening the model and verifying stronger progress properties.

6.4 Verifying the Protocol

In verifying that our model of the protocol satisfied the specification, we used the following strategy:

1. **Start with small combinations of caches and memories, and work up to the more complex hierarchical configurations.**

2. **Concentrate first on the simple safety properties given by formulas 6.1 through 6.4. These properties are all have the form of $\mathbf{AG}\, p_i$, where p_i is a propositional formula.**

The motivation behind the first element above is obvious. Why did we start with the simple safety properties? The idea is that we can check all of these properties using one forward search of the state space, checking at each step whether the set of states reached intersects the states satisfying $\neg p_i$ for any i. Once we have found a violation, we can terminate the search immediately and trace back to find a sequence of steps leading to the error. The ability to terminate the search early was important since the BDD representing the set of reached states tended to become very large once an erroneous transition had occurred. This is a fairly common phenomenon in BDD-based verification. In a correct system, there is often a nice characterization of the set of legal states, and this regularity is captured well by the BDDs. However, when the system is started outside of this set of states, it tends to make random-looking transitions, with the result that all regularity is quickly lost. By modifying SMV to perform this type of forward search with early termination, we saved a lot of time when doing the initial debugging. Once we had a model that satisfied all of the basic safety properties, we checked the more complex formulas (6.5 through 6.8). When evaluating the fixed points for the subformulas inside the **AG**, we restricted the searches to the set of reachable states. (As above, the idea was to avoid searching in ill-behaved parts of the state space.)

Even with the numerous simplifications made so far, verifying hierarchical configurations or configurations with more than a few processors required long execution times. For example, the very simple example of a single bus with two caches required about 10 minutes of CPU time on a Sun 3/60 to verify. In order to overcome this problem, we modified SMV so that we could use hiding and the *collapse* mapping to simplify the model. We then designated certain state components as hidden, and the restriction and collapsing were performed automatically. When verifying the properties discussed in the previous section, we hid all of the state components except:

1. the requester and responder attributes;
2. the cache line state and contents components; and
3. the error detection state components.

Note that all of the properties could be specified in terms of the above state components. With hiding and collapsing, the single-bus, two-cache example requires only a minute of CPU time. Table 6.1 shows the verification time (in CPU seconds) and BDD nodes required for single-bus configurations with two through seven caches. The TR BDD column shows the size of the BDD representing the transition relation, and SS BDD is the largest state set BDD. Both transition relation and state set BDDs grow linearly with the number of caches. The verification time grows roughly quadratically. (We also checked larger configurations with up to three buses and nine processor caches using a Sun 4. The state set BDDs grow linearly with the number of buses. The transition relation BDDs could grow linearly as well, but actually grow quadratically due to the way SMV represents transition relations.)

Caches	CPU time	TR BDD	SS BDD
2	60	7231	325
3	120	24834	715
4	225	50914	1128
5	345	79173	1541
6	535	107473	1954
7	870	135773	2367

Table 6.1: Verification times for single-bus configurations

6.5 Errors Discovered

Performing the verification exposed two errors in the standard. The first of these can actually occur in simple single bus configurations, which was somewhat surprising. Consider the system shown in figure 6.3. Initially, both caches are *invalid*. Processor P1 obtains an

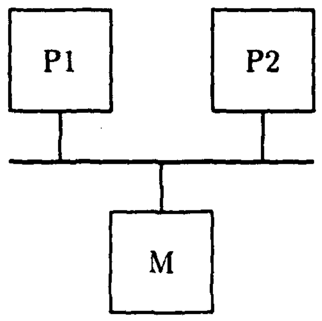


Figure 6.3: System exhibiting first error

exclusive-unmodified copy of the cache line. Next, P2 decides to issue a *read-modified*, which P1 splits for invalidation. However, the memory M does not split for access, and it supplies a copy of the cache line to P2. Under these circumstances, the standard specifies that P2 transitions to the *shared-unmodified* state. However, P1 does not acquire the *responder-invalidate* attribute. Instead, P2 is supposed to issue a subsequent *invalidate* to eliminate the copy of the line in P1's cache. Further, P1 retains an *exclusive-unmodified* copy of the line. This is obviously a dangerous situation, for now P1 can transition to *exclusive-modified* and write to the line before P2 issues the *invalidate*:

> A *CACHE* or *CACHE_AGENT* may set *EXCLUSIVE_MODIFIED* and clear *EXCLUSIVE_UNMODIFIED* if *EXCLUSIVE_UNMODIFIED*.
>
> A *CACHE* or *CACHE_AGENT* may allow read or modify access to the data in a cache line if *EXCLUSIVE_MODIFIED* is set.

The problem can be fixed by requiring that the processor cache P1 transition to the *shared-unmodified* state when it splits the *read-modified* for invalidation. There is a related problem when a *read-modified* is split for both access and invalidation. The proposed change eliminates the error in all situations.

The second error arises in hierarchical configurations such as the one shown in figure 6.4. P1, P2, and P3 all obtain *shared-unmodified*

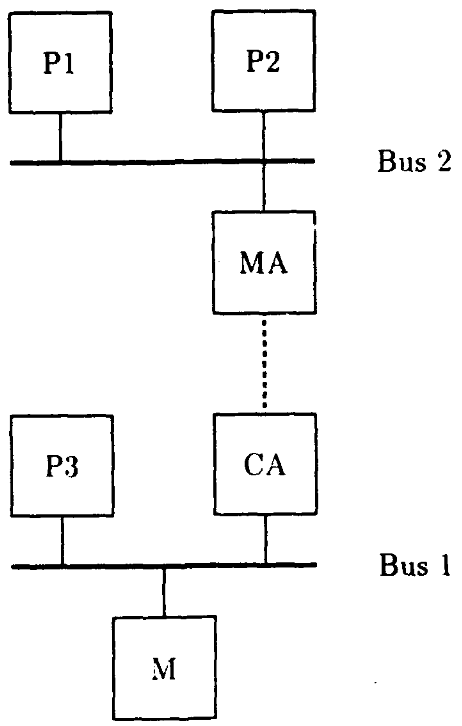


Figure 6.4: System exhibiting second error

copies of the cache line. P1 issues an *invalidate* transaction that P2 and MA split. P3 issues an *invalidate* that CA splits. The bus bridge detects that an invalidate-invalidate collision has occurred. That is, P3 is trying to invalidate P1, while P1 is trying to invalidate P3. Recall that in this situation, the standard specifies that the collision should be resolved by having the memory agent invalidate P1. When the memory agent tries to issue an invalidate for this purpose, P2 sees that there is already a transaction in progress for this cache line and asserts the *WT* signal on the bus.

> A module shall set *WAIT_CACHED* while
> *CACHED* ∧ *RESPONDER* ∧ (*READ_SHARED* ∨
> *READ_MODIFIED* ∨ *INVALIDATE* ∨ *READ_INVALID*).

MA observes this and acquires the ***requester-waiting*** attribute.

> A module shall set *REQUESTER_WAITING* if
> *MASTER* ∧ *WAIT_STATUS* ∧ (*READ_SHARED* ∨
> *READ_MODIFIED* ∨ *INVALIDATE* ∨ *READ_INVALID*).

Recall that when a module has this attribute, it will wait until it sees a completed response transaction before retrying its command. P2 now finishes invalidating and issues a ***modified-response***. Since P3 is not in the ***invalid*** state, this response must be split by MA. However, MA still maintains the ***requester-waiting*** attribute.

> A module shall clear *REQUESTER_WAITING* if
> *CACHED* ∧ (*SHARED_RESPONSE* ∨
> *MODIFIED_RESPONSE* ∧ ¬*SPLIT_STATUS* ∨
> *WRITE_INVALID*).

At this point, MA will not retry its command since it is still waiting for a completed response. However, no such response can occur; we have reached a deadlock. The deadlock can be avoided by having MA clear the ***requester-waiting*** attribute when it observes that P2 has finished invalidating. (It does this as follows: Caches assert *TF* when they split a ***modified-response***. The memory agent asserts *SR* for each ***modified-response*** to keep P1 from proceeding. When MA observes a ***modified-response*** with *TF* not asserted, it knows that all caches other than P1 are ***invalid***. It then clears ***requester-waiting*** and issues its ***invalidate***.)

6.6 Verifying Liveness

While the properties specifying absence of deadlock (6.6–6.8) were useful in finding errors, as we noted earlier, they are not preserved when we move to a different level of abstraction. In this section, we show how to prove stronger progress properties. We will concentrate on showing that if a cache issues a *read-shared* transaction, then eventually it obtains a readable copy of the cache line.

$$\mathbf{AG}(P.\mathit{master} \wedge P.\mathit{cmd} = \mathit{read\text{-}shared} \wedge \neg \mathit{WT} \rightarrow \mathbf{AF}\, P.\mathit{unmodified}) \tag{6.9}$$

Our original model in fact does not satisfy this specification. This is for the following reasons:

1. The arbitration model is unfair; a device may never have a chance to issue a command.
2. A module that owes a response to a split command may never issue the response, even if it is infinitely often the bus master.
3. In hierarchical configurations, the selection of which bus will next transition is unfair.
4. If a cache agent splits a command, the correspdonding memory agent may never pass on that command. (Similarly, a cache agent may not pass on commands split by the corresponding memory agent.)

In order to check the above property, we first had to strengthen our model. SMV provides a method for specifying acceptance conditions, and we used this facility. To ensure that arbitration is fair, we can just require that infinitely often, each device is chosen as the bus master. We can require that responses eventually be issued by enforcing

$$(\mathbf{GF}\, P.\mathit{master}) \rightarrow \mathbf{GF}(P.\mathit{responder} \neq \mathit{shared} \vee P.\mathit{master} \wedge P.\mathit{cmd} = \mathit{shared\text{-}response}).$$

(Unfortunately, SMV only supports fairness constraints of the form $\bigwedge_i \mathbf{GF}\, p_i$. However, we are already requiring that $\mathbf{GF}\, P.\mathit{master}$, so we simplified the above constraint to just

$$\mathbf{GF}(P.\mathit{responder} \neq \mathit{shared} \vee P.\mathit{master} \wedge P.\mathit{cmd} = \mathit{shared\text{-}response}).$$

This is known as the B^3 method [61].) To eliminate the last problem, we added some interlocks to the state of the bus bridge. When the cache agent splits a ***read-shared***, it eventually sets an interlock to let the memory agent knows that it needs a valid copy of the line. The memory agent sees this interlock and, if necessary, issues a ***read-shared*** on its bus to obtain the data. It also splits all requests that would require invalidating the data while the interlock is set. Eventually, the cache agent gets the data, issues a ***shared-response***, and clears the interlock.

We begin by considering just the single bus case. When we tried to verify property 6.9 directly, we found that the time and space required was excessive. One reason was that we could no longer hide most of the state components (as described in section 6.4). This is because the property and the acceptance conditions depend on the previously hidden components. Also, evaluating the **AF** operator when there are acceptance conditions requires a nested fixed point computation. A large number of iterations were needed for this computation to converge. Because of these problems, we did an assume-guarantee style verification. To begin, consider why we expect the property to be true, i.e., what properties of the environment of a cache must hold? If the environment does not split the ***read-shared***, then the cache will obtain the data as part of the transaction. If the ***read-shared*** is split, then the environment must eventually issue a *shared-response*:

$$\mathbf{AG}(P.master \wedge P.cmd = read\text{-}shared \wedge \neg WT \wedge SR \\ \rightarrow \mathbf{AF}\ CMD = shared\text{-}response). \quad (6.10)$$

(CMD is the command line on the bus.)

We now use this property as an assumption about the environment of P and then try checking the desired specification. It does not hold; the counterexample produced by the model checker shows a situation in which the environment behaves so as to cause the $P.error$ state component to become true. As we have already verified

$$\mathbf{AG}(\neg d.bus\text{-}error \wedge \neg d.error)$$

(property 6.1), we make this an assumption as well. The verification again fails, and the trace shows a situation where the ***read-shared*** is not split, but where the cache fails to transition to a readable state because *CMD* is not equal to *P.cmd*. This obviously should not occur, so we add the following assumption:

$$\mathbf{AG}(P.master \rightarrow CMD = P.cmd). \qquad (6.11)$$

With the above assertion, the property still may not hold: if the environment asserts *WT* while issuing the *shared-response*, the processor cache does not change its state. However, modules should not assert *WT* in this situation:

$$\mathbf{AG}(CMD = \textit{shared-response} \rightarrow \neg WT). \qquad (6.12)$$

Taken together, these assumptions are strong enough to imply the property 6.9. At this point, we have verified

$$\langle 6.10, 6.1, 6.11, 6.12\rangle P \langle 6.9\rangle.$$

Of the assumptions that needed to be discharged, 6.10 is the most complex, so we consider it first. We will check it using the bus model *B* plus some assumptions about the devices on the bus. The natural assumption about each device is:

$$\mathbf{AG}(\neg d.master \wedge CMD = \textit{read-shared} \wedge \neg WT \wedge sr \\ \rightarrow \mathbf{AF}\; CMD = \textit{shared-response}). \qquad (6.13)$$

This states that if a device splits a *read-shared*, then eventually a *shared-response* must occur. Making this assumption about every device (except *P*) and then checking 6.10 shows that 6.10 did not hold. The error trace involves *P* splitting its own *read-shared*. This is clearly illegal, so we assume

$$\mathbf{AG}(P.master \wedge P.cmd = \textit{read-shared} \rightarrow \neg P.sr). \qquad (6.14)$$

With this additional assumption, property 6.10 is true.

$$\langle 6.14, 6.13 \text{ for each } d\rangle B \langle 6.10\rangle$$

To verify 6.13, we go back to the model of a processor cache. Based on our earlier experience, we assume that no errors would be detected and that any commands the cache issues will appear on the bus (properties 6.1 and 6.11). We also assume that the cache will get to be bus master to issue the ***shared-response***:

$$\mathbf{AG\,AF}\ \mathit{P1.master}. \tag{6.15}$$

With these assumptions, the model checker produces an error trace that shows the cache splitting a ***read-shared*** and the environment subsequently issuing a ***read-modified***. The cache raises *P1.wt* to hold up this read, but *WT* does not go high on the bus. At this point, the P1 loses the *responder-shared* attribute. We therefore assume

$$\mathbf{AG}(\mathit{P1.wt} \rightarrow \mathit{WT}). \tag{6.16}$$

This is still not sufficient to prove 6.13. The counterexample has P1 observing *WT* on the bus and acquiring the ***requester-waiting*** attribute before splitting the ***read-shared*** and acquiring ***responder-shared***. In this case, the device that raises *WT* should also assert it during the *read-shared*. In general, P1 should never have nontrivial requester and responder attributes at the same time:

$$\mathbf{AG}(\mathit{P1.responder} = \mathit{none} \vee \mathit{P1.requester} = \mathit{none}). \tag{6.17}$$

Adding this assumption is sufficient:

$$\langle 6.1, 6.11, 6.15, 6.16, 6.17\rangle P1 \langle 6.13\rangle.$$

A similar proof shows that memory also satisfies 6.13 (with slightly weaker assumptions).

Property 6.17 can be verified using the method described in section 6.4. Basic properties such as 6.15 and 6.16 are checked using just the bus model *B* with no assumptions. The fact that caches do not split their own *read-shared* commands (6.14) can be verified using just the processor cache model. To show that *WT* is not asserted during *shared-response* transactions (6.12), we show that for each device

$$\mathbf{AG}(\mathit{CMD} = \mathit{shared\text{-}response} \rightarrow \neg \mathit{d.wt}),$$

and then used these properties as assumptions together with the bus model. At this point, all assumptions have been discharged. About two minutes of CPU time were required to verify all of the assumptions except 6.1 and 6.17. (The time required for these is given in table 6.1.)

Let us now consider a hierarchical configuration (figure 6.5). Suppose that we want to prove property 6.9 for P3. The key point is to demonstrate that CA always responds to those *read-shared* transactions that it splits (property 6.13). We expect this to be true since:

1. the system of interlocks that we added to the model will cause MA to issue a *read-shared* if needed; and

2. we should be able to prove analog of property 6.9 for *read-shared* transactions issued by the memory agent.

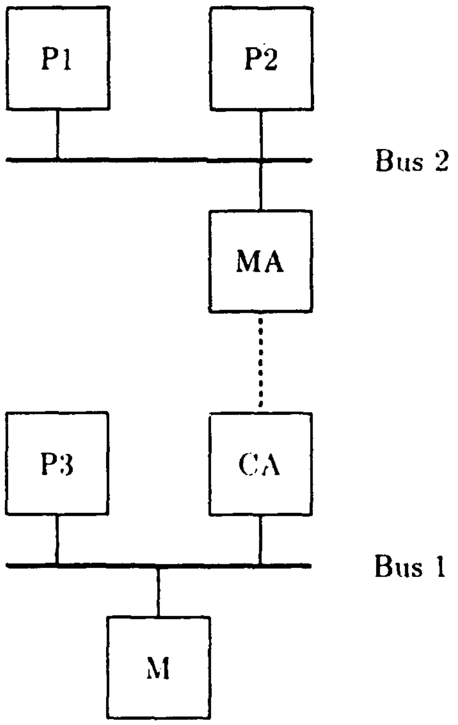


Figure 6.5: Hierarchical configuration

To try to prove 6.13 for CA, we used the cache agent and memory agent models. We assumed the latter condition above, plus other basic properties such as 6.1 and 6.11. Several iterations were devoted to getting the interlock mechanism working. Even after this however, the property was still found to be false. Based on the error trace, we concluded that 6.9 in fact does not hold in hierarchical systems. The scenario indicated by the counterexample is the following (refer to figure 6.5). P1 obtains a *exclusive-modified* copy of the cache line. At this point, the bridge is in the *remote-exclusive-modified* state. P3 issues a *read-shared* on the bottom bus, and CA splits the transaction. An interlock is set to tell the memory agent to retrieve the data. P2 issues a *read-modified* on the top bus, and P1 intervenes and splits the transaction. Next, MA sees the interlock and tries to issue a *read-shared*. However, since P1 is already processing a split transaction for the line, it asserts *WT* and MA acquires the *requester-waiting* attribute. Now P1 issues a *modified-response* and transitions to the *invalid* state, while P2 goes to *exclusive-modified*. MA clears the *requester-waiting* attribute, but before it can rearbitrate for the bus, P1 issues a *read-modified* which P2 intervenes in and splits. At this point the process repeats: MA tries to issue a *read-shared* but is told to wait by P2, and eventually the modified cache line passes back to P1. Thus, MA never successfully issues the *read-shared* and never obtains the data. What we have found is that fair arbitration is not sufficient to guarantee absence of livelock. (Actually, it is hard to imagine any arbitration scheme that would avoid this problem. It seems that some sort of queue-based system for recording requests would be required. Because this represents a substantial change to the protocol, we did not attempt to develop a model that guarantees progress. Another possibility would be to require that the memory agent be sufficiently fast. That is, if the memory agent is guaranteed to rearbitrate immediately to try to issue the *read-shared* after it sees the *modified-response*, and if the arbitration is fair, then it may be possible to prove progress.)

Overall, our approach of performing assume-guarantee style verification by working backwards from the desired property seems to be fairly natural. Counterexamples from the model checker are used to guide the selection of appropriate assumptions at each stage. Further, in situations like the above where the property that we are trying to verify does *not* hold, we are eventually led to a counterexample representing a real error condition. This is an important point, since most of the verification time and effort is spent working on incorrect designs. Assume-guarantee reasoning can still be an effective tool in these situations.

6.7 Summary

We have demonstrated our verification techniques using a substantial example, the IEEE Futurebus+ cache coherence protocol. We constructed an abstract model of the protocol and checked whether it satisfied a temporal logic specification of cache coherence. In performing the verification, we found two errors. We used assume-guarantee reasoning to check liveness properties of the protocol. We were able to show that the single-bus version of the protocol did satisfy our liveness specification, but that livelocks may occur in a hierarchical configuration.

Chapter 7

Conclusion

We have described methods for doing compositional verification and for using abstraction in the context of temporal logic model checking. Our techniques are based on ACTL, a subset of CTL in which we eliminate the **E** path quantifier. We showed how to do full assume-guarantee style reasoning with ACTL, and how to use abstraction to verify systems that manipulate data in non-trivial ways. To demonstrate that our techniques were practical, we used abstraction and assume-guarantee style reasoning to verify the IEEE Futurebus+ cache coherence protocol. During the verification process, we discovered errors in the IEEE standard.

While we have considered a number of examples besides the Futurebus+ protocol, we would like to gain more experience in trying to apply our techniques to real systems. We feel that it is particularly important to look at a single system across several levels of abstraction. Recall that in the Futurebus+ example, we constructed the abstract model directly since we did not have a formal low-level model. (While much of the standard is expressed in terms of boolean attributes, they are poorly structured, not entirely formal, and incomplete.) As such, we were not able to automatically apply abstraction via observers or the techniques described in section 4.2 to this example. It would be interesting to develop an implementation-level description of one of the types of Futurebus+ modules (e.g., a processor board) and to try to show that the model we used is a valid abstraction.

There are also a number of theoretical questions that we would like to address. One question concerns the exact complexity of the compositional model checking problem for full CTL. In chapter 2, we showed that it was NP-hard, and, since our interest is mainly in practical methods, developed a polynomial-time approximation algorithm. However, it is not even entirely clear that the problem is decidable. One approach for showing decidability would be to try to reduce the problem to a containment problem on tree automata. It is well known that the set of computation trees satisfying a CTL formula is an ω-regular tree language [46, 81] and is accepted by a finite automata on infinite trees. If the set of computation trees representing the closed systems that can be obtained by composition with a given Moore machine is also ω-regular, then the problem can be solved by testing inclusion between the two automata.

Another problem involves deciding whether $\preceq$ holds between two arbitrary structures. We believe that we have a polynomial-time algorithm for this problem, but we have not proved it correct. It roughly involves executing a fixed point computation like that involved in testing language inclusion as discussed by Clarke, Draghicescu, and Kurshan [26]. However, even if our algorithm is correct, we are not optimistic that it will work well in a BDD-based setting. It may be more important to look for additional approximation algorithms for this problem.

We believe that it may be possible to use ideas similar to those in section 4.2 in order to generate abstractions of infinite state systems. If the program describing the system is written in terms of abstract data types described by algebraic specifications [65], we believe that automated theorem proving and term rewriting techniques could be used to derive abstract versions of the primitive operators. Of course the abstracted systems would have to be finite state in order to apply our model checking tools. It is not yet clear how conservative these finite approximations will be.

Finally, there is work to be done on helping the user apply the techniques that we have proposed. Our methods require that the user decide on appropriate assumptions during an assume-guarantee proof and on what abstractions to make. It is unlikely that either of these steps can be fully automated, but it may be possible to provide hints. For example, if the user is trying to verify a system involving a data path, then the examples that we verified in chapters 4 and 5 can suggest useful abstractions. There is also the problem of providing feedback when the verification fails. Since our verification methods are conservative, the traces produced by the tools when verifying an abstract model need not actually correspond to legal executions in the actual system. It may be possible to use the information from the abstract-level verification to constrain a lower-level search for an actual error trace. Alternatively, lower-level information might be used to guide the generation of a meaningful abstract-level counterexample.

Appendix A

Summary of BDDs

Reduced ordered binary decision diagrams (BDDs) are a canonical form representation for boolean functions [17]. They are often substantially more compact than traditional normal forms such as conjunctive normal. form and disjunctive normal form, and they can be manipulated very efficiently. Hence, they have become widely used for a variety of applications, including symbolic simulation, verification of combinational logic, logic synthesis, and finite-state verification. A BDD is similar to a binary decision tree, except that its structure is a directed acyclic graph rather than a tree, and there is a strict total order placed on the occurrence of variables as one traverses the graph from root to leaf. Figure A.1 shows an example BDD. It represents the function $(a \wedge b) \vee (c \wedge d)$, using the variable ordering $a < b < c < d$. Given an assignment of boolean values to the variables a, b, c and d, one can decide whether the assignment makes the function true by traversing the graph beginning at the root and branching at each node based on the value assigned to the variable that labels the node. For example, the assignment $\{a = 1, b = 0, c = 1, d = 1\}$ leads to a leaf node labeled 1, hence the function is true for this assignment.

Bryant showed that, given a variable ordering, there is a canonical BDD for every function [17]. This canonical form is obtained by starting from an ordered (but not necessarily reduced) binary decision diagram and applying the following two reduction rules. First, if two nodes n_1 and n_2 in the graph are isomorphic, then we delete n_2 and redirect all of the arcs going into n_2 so that they point to n_1. Second, if the two arcs

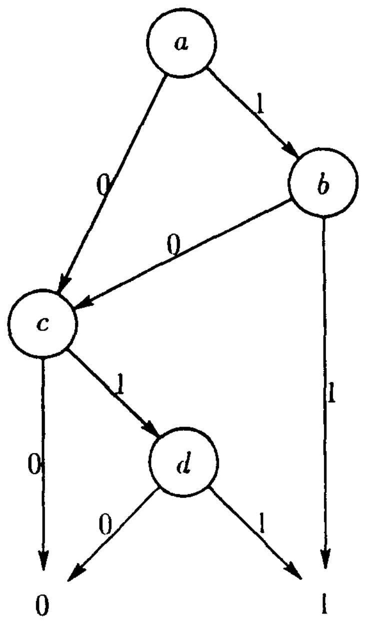


Figure A.1: A BDD representing $(a \wedge b) \vee (c \wedge d)$

coming out of a node n_1 both point to the same node n_2, then we delete n_1 and redirect all arcs into n_1 so that they point to n_2. Eventually, we will not be able to apply either rule. At this point, the graph is a canonical reduced ordered binary decision diagram. The size of a BDD can depend critically on the variable ordering. For many of the functions that seem to arise in practice, there are orderings for which the BDD size is polynomial in the number of variables.

An important property of BDDs is that they degrade gradually: most operations can be performed in polynomial time, and the results of those operations are polynomial in the size of the inputs to the operation. Given BDDs for f and g, logical combinations of these functions such as $f \vee g$ and $\neg f$ can be computed in time linear in the product of the sizes of the argument BDDs. Quantification over a single variable ($\exists x.f$) is requires polynomial time, but quantification over a set of variables may be exponential in the number of variables. However, in practice, quantification is usually efficient since it reduces the number of variables that the function depends on. Substituting a function g for the variable x in f is also polynomial. Multiple (simultaneous) substitution is exponential in the worst case, but again is usually well-behaved in practice.

In our work, we use BDDs as a means of representing sets, relations, and functions over finite domains, and for manipulating these objects. Given a finite domain D, we first encode the elements of D using a set V of boolean variables. Let us suppose for simplicity that D has exactly 2^k elements and that V consists of k variables. Then every valuation of the variables in V corresponds to exactly one element of D. A boolean function χ over V can be identified with the set of valuations that make the function true. By identifying each such valuation with the corresponding element of D, we can view χ as representing a subset of D. This is called the *characteristic function representation* of the set. Relations over, e.g., $D \times D$ can be represented in a similar way, except now we need $2k$ boolean variables to encode the pairs of elements of D. Functions are simply viewed as a special case of relations. There is a close correspondence between set and relational operations and logical operations on the corresponding characteristic functions. For example, if D_1 and D_2 are subsets of D and are represented by χ_{D_1} and χ_{D_2}, respectively, then the characteristic function for $D_1 \cup D_2$ is simply $\chi_{D_1} \vee \chi_{D_2}$. Other operations that can be performed efficiently include intersection, quantification over elements of D, functional and relational composition, etc.

Bibliography

[1] F. Van Aelten, S. Y. Liao, J. Allen, and S. Devadas. Automatic generation and verification of sufficient correctness properties for synchronous processors. In *Proceedings of the 1992 IEEE International Conference on Computer-Aided Design.* IEEE Computer Society Press, November 1992.

[2] S. A.Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.

[3] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[4] D. L. Beatty, R. E. Bryant, and C.-J. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference.* IEEE Computer Society Press, June 1991.

[5] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[6] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G. V. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth Workshop on Computer-Aided Verification*, July 1992.

[7] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[8] G. V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3), March 1982.

[9] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In L. Claesen, editor, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, November 1989.

[10] A. Bouajjani, J. C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–271, June 1992.

[11] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1990.

[12] J. Bradfield and C. Stirling. Local model checking for infinite state spaces. In K. G. Larsen and A. Skou, editors, *Proceedings of the Third Workshop on Computer-Aided Verification*, July 1991.

[13] M. C. Browne. An improved algorithm for automatic verification of finite state machines using temporal logic. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1986.

[14] M. C. Browne. *Automatic verification of finite state machines using temporal logic*. PhD thesis, Carnegie Mellon University, 1989.

[15] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[16] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2), July 1988.

[17] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[18] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[19] J. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11:481-494, 1964.

[20] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science*. Stanford University Press, 1962.

[21] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, 1992.

[22] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1991.

[23] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1990.

[24] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142-170, June 1992.

[25] L. Claesen, F. Proesmans, E. Verlind, and H. De Man. A methodology for the automatic verification of MOS transistor level implementations from high level behavioral specifications. In *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.

[26] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N. D. Jones, editors, *Proceedings of the 15th Colloquium on Trees in Algebra and*

Programming, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1990.

[27] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[28] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[29] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[30] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[31] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1989.

[32] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9):1283–1292, September 1991.

[33] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[34] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third ACM Symposium on the Theory of Computing*, 1971.

[35] F. Corella. Automated high-level verification against clocked algorithmic specifications. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[36] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. Claesen, editor, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, November 1989.

[37] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[38] O. Coudert, C. Berthet, and J. C. Madre. Verifying temporal properties of sequential machines without building their state diagrams. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[39] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the 1990 IEEE International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1990.

[40] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, January 1977.

[41] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, January 1979.

[42] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. Submitted for publication.

[43] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.

[44] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, Part E 133(5), 1986.

[45] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, June 1986.

[46] E. A. Emerson and A. P. Sistla. Deciding full branching time logic. *Information and Control*, 61:175–201, 1984.

[47] T. Filkorn. Functional extension of symbolic model checking. In K. G. Larsen and A. Skou, editors, *Proceedings of the Third Workshop on Computer-Aided Verification*, July 1991.

[48] T. Filkorn. A method for symbolic verification of synchronous circuits. In D. Borrione and R. Waxman, editors, *Proceedings of the Tenth International Symposium on Computer Hardware Description Languages and their Applications.* North-Holland, April 1991.

[49] M. Fujita. RTL design verification by making use of datapath information. In *Proceedings of the 1992 IEEE International Conference on Computer Design.* IEEE Computer Society Press, October 1992.

[50] P. Godefroid. Using partial orders to improve automatic verification methods. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[51] S. Graf and B. Steffen. Compositional minimization of finite state processes. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[52] O. Grumberg and D. E. Long. Model checking and modular verification. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1991.

[53] Z. Har'El and R. P. Kurshan. The COSPAN user's guide. Technical Report 11211-871009-21TM, AT&T Bell Laboratories, 1987.

[54] Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan.-Feb. 1990.

[55] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[56] M. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[57] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[58] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[59] IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*, March 1992. IEEE Standard 896.1-1991.

[60] B. Josko. Verifying the correctness of AADL-modules using model checking. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1989.

[61] T. Kidder. *The Soul of a New Machine*. Avon, 1982. B^3: "Big Brown Bag".

[62] R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Stepwise Refinement of Distributed*

Systems, Models, Formalisms, Correctness, volume 430 of *Lecture Notes in Computer Science.* Springer-Verlag, May 1989.

[63] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing.* ACM Press, August 1989.

[64] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985.

[65] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[66] Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H. T. Kung, B. Sproull, and G. Steele, editors, *VLSI Systems and Computations.* Computer Science Press, 1981.

[67] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[68] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the 1991 International Symposium on Shared Memory Multiprocessors*, April 1991.

[69] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045 1079, 1955.

[70] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the Second Internation Joint Conference on Artificial Intelligence*, September 1971.

[71] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[72] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*. Princeton University Press, 1956.

[73] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

[74] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[75] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, 1977.

[76] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[77] A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series. Series F, Computer and system sciences*. Springer-Verlag, 1984.

[78] A. Pnueli and R. Sherman. Semantic tableau for temporal logic. Technical Report CS81-21, The Weizmann Institute, 1981.

[79] D. K. Probst and H. F. Li. Using partial order semantics to avoid the state explosion problem in asynchronous systems. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[80] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[81] M. O. Rabin. Decidability of second order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[82] N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, 1971.

[83] C. L. Seitz. Ideas about arbiters. *Lambda*, 10(4), 1980.

[84] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[85] G. Shurek and O. Grumberg. The modular framework of computer-aided verification: Motivation, solutions and evaluation criteria. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[86] C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. In J. Diaz and F. Orejas, editors, *Proceedings of the 1989 International Joint Conference on Theory and Practice of Software Development*, volume 351–352 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1989.

[87] R. S. Streett. A propositional dynamic logic of looping and converse. *Information and Control*, 54:121–141, 1982.

[88] A. Tarski. A lattic-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[89] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Proceedings of the 1990 IEEE International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1990.

[90] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, 1989.

[91] A. Valmari. A stubborn attack on the state explosion problem. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[92] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1986.

[93] G. Winskel. Compositional checking of validity on finite state processes. Draft copy.

[94] G. Winskel. Model checking in the modal ν-calculus. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, 1989.

[95] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[96] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, January 1986.

[97] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.